Recovery Support for Real-time Distributed Editing Systems

Mohammed I. Alghamdi[1], Xunfei Jiang[2], Ji Zhang[3], Jifu Zhang[4], Xiao Qin[3]

[1] Department of Computer Science, Al-Baha University, Kingdom of Saudi Arabia
[2] Department of Computer Science, Earlham College, USA
[3] Department of Computer Science and Software Engineering, Auburn University, USA
[4] School of Computer Science and Technology, Taiyuan University of Science and Technology, China
mialmushilah@bu.edu.sa, jiangxu@earlham.edu, jizhang@auburn.edu, jifuzh@sina.com, xqin@auburn.edu

Abstract

Crash recovery techniques allow real-time distributed editing systems to make progress in case of failures. In this study, we propose a recovery scheme to manage a local document state (a.k.a., checkpoint) in each node, which periodically generates the checkpoint state. If a transient failure occurs in a distributed editing system, a node can rejoin the editing system by loading the local document state rather than retrieving the state from remote nodes. Our recovery scheme maintains the consistency between a local state and a remote state during the crash recovery procedure. The correctness of the recovery algorithm is theoretically proved. We evaluate the performance of our recovery scheme by varying the elapsed time between a failed node joining and leaving a system. The experimental results show that our solution is superior to the traditional recovery approach that regains document states from other peer nodes.

Keywords: Distributed computing, Real-time systems, System recovery

1 Introduction

Distributed real-time editing systems enable a group of geographically distributed users to simultaneously view and edit shared documents [3, 17, 24-25, 27]. Important features of a distributed editing system include quick responsiveness, supporting unconstrained collaboration, and tolerant failed processes. A distributed system should allow nodes to freely rejoin the system after any node or link failures, allowing users at functioning nodes to continue their editing work and failed nodes rejoin a group at any time.

It is indispensable for a distributed real-time editing system to tolerate node and link failures [19]. Two commonly adopted fault-tolerant techniques include replication [7, 16] and persistence [20]. In a replication scheme, hardware and software components redundantly process the same messages in the same order. In case of a failure of any component, the other components are still able to continue processing tasks. Persistence-based techniques rely on checkpointing whereby during the normal execution, system states are periodically saved on a stable storage; checkpoints will be retrieved during a crash recovery process to rollback to an earlier consistent state.

It is a traditional wisdom that the recovery of a failed node is implemented through regaining system document states from other surviving nodes. The downside of retrieving document states from remote nodes is that recovery latency becomes significantly long if document state data is huge. Delays may be substantially reduced when there is no need to start the recovery from scratch. A failed node may rejoin a distributed editing system without starting from the very beginning if an appropriate checkpoint can be locally loaded.

In this study, we investigate a new crash recovery approach to maintaining a local document state in each node, which periodically generates document checkpoints. In doing so, if a failure occurs in a node or network connections, the node is capable of rejoining the editing system by loading its local document state rather than obtaining the document checkpoint from remote nodes. During the recovery procedure, a recovery algorithm is responsible for maintaining the consistency between a local state and a remote state.

In this paper, we propose a crash recovery scheme for distributed real-time editing systems by managing local document states or checkpoints of each node. Checkpoints are stored on permanent storage in nodes. We focus on distributed editing systems without any centralized server; therefore, there is need to provide a fault-tolerant support for centralized servers [27].

If a node fails due to transient errors (e.g., disconnections from a distributed system), the node is able to rejoin the editing system by loading document checkpoints. To synchronize with the system's current document state, other peer nodes resend necessary editing operations based on the loaded checkpoints. In

*Corresponding Author: Xunfei Jiang; E-mail: jiangxu@earlham.edu
DOI: 10.3966/160792642018081904015

this study, we pay attention to editing operations that should be resent by all the other nodes. We describe a system model and the crash recovery algorithms, the correctness of which is theoretically proved. We conduct extensive experiments to demonstrate that our crash recovery approach outperforms conventional recovery solutions that regain document states from other nodes in a distributed system.

The rest of the paper is organized as follows. In Section 2, we present the system model of distributed editing systems. The crash recovery algorithms can be found in Section 3. The performance evaluation of our novel crash recovery is outlined in Section 4. Section 5 summarizes the related work. Section 6 concludes this paper with future directions.

2 Problem Formulation

We model a real-time distributed editing system as a pair, i.e., DES=<S, C>, where S is a finite set of nodes $S=\{s_1, s_2, ..., s_n\}$. Node s_i is a node involved in editing work. C is a finite set of channels, i.e., $C = \{c_{ij}, 1 \leq i \leq n, i<j \leq n\}$, where c_{ij} is a point-to-point channel connecting node s_i and node s_j in a distributed system. s_i's execution is represented in form of a sequence of editing operations, including remote operations issued from other nodes in the system. LDS_i denotes the local document state of node s_i, which periodically generates and stores the states on permanent storage. In case of the transient failures of node s_i, $LDS_i$is loaded to quickly initialize the node.

In what follows, we formally define editing operations, execution forms, and execution times.

Definition 1. Given an operation O, then $s(O)$ denotes the node at which O is generated, $e_i(O)$ represents the execution form of O at s_i, $gt_i(O)$ denotes the time when s_i generates O, and $at_i(O)$ represents the execution time of O at the remote site s_i. It is certain that $gt_i(O) \rightarrow s(O) = i$, and $at_i(O) \rightarrow s(O) \neq i$.

The above definition illustrates two implications. First, $gt_i(O)$ implies that the node of operation O is node i (i.e., $s(O) = i$). Second, $at_i(O)$ suggests that the node of operation O is not node i (i.e.$s(O) \neq i$).

We define the causal order between two operations below. It is worth mentioning that the causal order is an important concept used to prove the correctness of our crash recovering approach.

Definition 2. Given two operations $O_i$andO_j, O_i is causal order preceding O_j, denoted by $O_i \mapsto O_j$, iff:

(1) $s(O_i) = s(O_j) = k$, and $gt_k(O_i) < gt_k(O_j)$; or

(2) $s(O_i) \neq s(O_j)$, $at_k(O_i) < gt_k(O_j)$, where $k = s(O_j)$; or

(3) There exists an operation O_k, such that $O_i \mapsto O_k$, $O_k \mapsto O_j$.

Definition 2 indicates that O_i is causal order preceding O_j if and only if one of the following three conditions is satisfied. First, if O_i and O_j are issued on the same node (e.g., node k), then O_i is issued earlier than O_j. Second, we consider a case where O_i and $O_j$are created on two different nodes. For example, O_j is created on node k and O_i is issued on a node other than node k. In this case, the arrival time of O_i on node k must be earlier than O_j's creation time (i.e. $gt_k(Oj)$). Third, there is a third operation (e.g., O_k) that has the causal order relations with $O_i$and O_j. Thus, O_i is causal order preceding O_k and O_k is causal order preceding O_j.

The definition below specifies the independent relation between two operations. Thus, two operations are independent of each other if there is no causal order relation between the two operations.

Definition 3. Operation O_i and $O_j$are independent if and only if neither $O_i \mapsto O_j$, nor $O_j \mapsto O_i$, which is defined as $O_i \| O_j$.

In what follows, Definition 4 introduces the concept of a context associated with an operation. The context concept is a determining factor for crash recovery overhead (see also Section 4).

Definition 4. An operation is associated with a context, denoted as CT_O, which is the list of operations that need to be executed to bring the document from its initial states to the states on which O is defined.

The definition below specifies the condition under which two operations are context equivalent.

Definition 5. Given two operations O_i and $O_j$associated with contexts CT_{O_i} and CT_{O_j}, O_i and O_j are context equivalent, i.e., $O_i : O_j$, if and only if $CT_{O_i} = CT_{O_j}$.

We define the two editing operations' relation in terms of context preceding below.

Definition 6. Given two operations O_i and O_j associated with contexts CT_{O_i} and CT_{O_j}, O_i is context preceding O_j, i.e., $O_i > O_j$, if and only if $CT_{O_j} = CT_{O_i} + [O_j]$.

The total-order relation between two operations is defined as follows.

Definition 7. We consider two operations O_i and O_j, $s(O_i) = a$, $s(O_j) = b$, and timestamped by SV_{O_i} and SV_{O_j}, respectively [25]. We say O_i is total order preceding O_j, (i.e., $O_i \Rightarrow O_j$), iff (1) sum(SV_{O_i}) < sum(SV_{O_j}) or (2) $a<b$ when sum (SV_{O_i}) = sum (SV_{O_j}), where sum $(SV) = \sum_{i=1}^{n} SV[i]$.

Definition 8. Let HB_i^t be the history buffer of s_i at time t. In history buffer HB_i^t, $y_i^{j,t}$ denotes the latest operation generated in node s_j, iff $\forall O \in HB_i^t$, $O \neq y_i^{j,t}$: $s(O) = j \rightarrow (O \mapsto y_i^{j,t})$.

In a distributed editing system, each node s_i maintains a status ξ_i, which can be one of the following six candidates, namely, *join*, *run*, *checkpoint*, *recovery*, *fail*, and *finish*. A crash recovery procedure begins by loading a local document state (a.k.a., document checkpoint) from the permanent storage to the crashed node. If no local document state is available, the state is initialized to join and remains in the *join* state until the node receives a remote document state from the other nodes and executes operations according to the remote state. In contrast, if the node keeps a local document state, then the setting up of this node relies on the local state. The local state of *finish* means that this node has successfully exited during the past session. In this case, the local state changes from *finish* into *join*; the node obtains the remote document state from the other nodes; the local state changes from *join* into *run* after the node starts executing operations according to the remote document state. In case the local document state's status is run, this node has not exited successfully due to a link failure or node failure. Hence, the state changes into *recover* followed by loading all data in the local document checkpoint. After the *finish* state, in which local document state is obtained and all missed operations entered at its own node are received, the state is set to *run*. The distributed editing system's user interface is not enabled until the status of the node becomes *run*. We formally describe the state transitions in a theorem.

Now we consider a case where the current state of a local node is *join*. The local node propagates a join message to all the other nodes in the editing system, then the node waits for the first remote node to reply this *join* message. After receiving the document state from this remote node and the node is initialized, the status of the node changes into run. If the local node does not receive any reply, the node simply assumes that it is the first one joining the system. In this case, a local document is loaded and the status of the local node is updated into run.

If the node's status is *checkpoint*, the node stores the local document state on its local permanent storage. After the document checkpoint has been made, the node's status is transitioned into *run*. In case that the status is *finish*, the node saves the local document checkpoint, followed by broadcasting the *finish* message to all the other nodes. Such a notification informs the other nodes that the local node has finished making a document checkpoint.

If an operation is a local *finish* operation, the node's status is switched from *run* into *finish*. If the operation is other types of local operations, the node executes the operation, appends the operation into its history buffer, and broadcasts the operation to the system's other nodes. If the operation is a remote operation originally issued at another remote node, the operation must be transformed before being locally executed (see details on operation transformation in [25]). The diagram of status transitions is depicted in Figure 1.

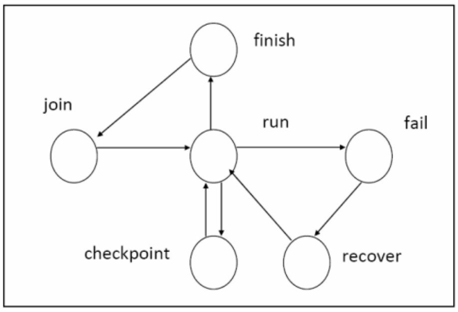

Figure 1. Diagram of status transitions

In a replicated scheme, concurrency control to maintain consistency in a replicated document is one of the key challenging issues. To solve the critical inconsistency problems, the consistency model addressed in our study has the following three vital properties [25]:

Convergence Property. When the same set of operations have been executed at all participating nodes, all copies of a shared document are identical.

Causality-preservation Property. We consider two operations O_i and O_j. If O_i is causal order preceding O_j (i.e., $O_i \Mapsto O_j$), then O_i executes before O_j at all nodes.

Intention-preservation Property. The effect of an operation O at remote nodes is the same as that of the operation at its local node at the time of its generation; the effects of independent operations do not interfere with each other.

3 Crash Recovery

Prior to the description of our crash recovery algorithm, we propose an algorithm to determine the latest operation generated at node s_j in HB_i^t below.

Given two local editing operations created at the same node, these operations satisfy three conditions, which are formally presented in the form of the following three lemmas. These lemmas not only clarify the relationships among locally generated operations but also help in proving our theorems (see Theorems 3.4-3.8).

Lemma 3.1. If two operations O_i and O_j are created at the same node (i.e., $s(O_i) = s(O_j)$), then either O_i is causal order preceding O_j (i.e., $O_i \Mapsto O_j$) or O_j is causal order preceding O_i (i.e., $O_j \Mapsto O_i$).

Proof. The proof of this lemma is straightforward and skipped.

Lemma 3.2. Given two operations O_i and O_j, if O_i is causal order preceding O_j (i.e., $O_i \Mapsto O_j$), then O_i is total order preceding O_j (i.e., $O_j \Mapsto O_i$). Thus, we have $\forall O_i, O_j$: $(s(O_i) \Mapsto s(O_j)) \rightarrow (O_i \Mapsto O_j)$.

Proof. Please refer to [21] for the proof of this lemma.

Lemma 3.3. Let us consider two operations O_i and O_j in the history buffer HB. If two operations are generated at the same node and O_i is total order proceeding to O_j (i.e., $O_i \Rightarrow O_j$), then O_i is causal order preceding O_j (i.e., $O_i \Mapsto O_j$). Thus, we have $\forall O_i, O_j \in$

HB: ($s(O_i) = s(O_j) \wedge O_i \Rightarrow O_j) \rightarrow (O_i \mapsto O_j)$.

Proof. We prove this lemma by contradiction. Assuming that lemma 3.3 is incorrect, we show that either O_j is causal order preceding O_i (i.e., $O_j \mapsto O_i$) or O_i and O_j are two independent operations (i.e., $O_i \parallel O_j$). Because these two operations are issued on the same local node (i.e., $s(O_i) = s(O_j)$), lemma 3.1 suggests that O_j is causal order preceding O_i (i.e., $O_j \mapsto O_i$). Thus, O_j is total order preceding O_i (i.e., $O_j \Rightarrow O_i$) (see lemma 3.2), which is a contradiction. This proves the lemma. □

Given HB_i^t and s_j, we design Algorithm 1 to determine the latest operation generated at node s_j. We prove the correctness of Algorithm 1 in the following theorem.

Theorem 3.4. Given HB_i^t and node s_j, Algorithm $LO(HB_i^t, j)$ determines that the latest operation issued at node s_j is y_i^j.

Proof. Because Algorithm $LO(HB_i^t, j)$ scans history buffer HB_i^t from right to left, we have $\forall O_k \in HB_i^t$, $O_k = HB_i^t[a]$, $O = HB_i^t[b]$, $O_k \neq O$: $s(O_k) = s(O) = j \rightarrow a < b$. Thus, it is proved that O_k is total order preceding O (i.e., $O_k \Rightarrow O$). Then, lemma 3.3 shows that O_k is causal order preceding O (i.e., $O_k \mapsto O$). Hence, we have $\forall O_k \in HB_i^t$, $O_k \neq O$: $s(O_k) = j \rightarrow (O_k \mapsto O)$. According to Definition 8, operation O is $y_i^{j,t}$ (i.e., $O = y_i^{j,t}$), which concludes the proof of the theorem. □

Algorithm 1. $LO(HB_i^t, j)$: Given a history buffer HB_i^t at node s_i, y_i^j is the latest operation generated at s_j; it is obtained as the follows

```
j ← | HB_i^t |;
while j > 0 do
    O ← HB_i^t [j];
    if s(O) = j then
        return y_i^j = O;
    else
        j ← j - 1;
    end if
end while
return φ;
```

When a link or a node fails, the node will be allowed to rejoin the editing system without starting from scratch. In our crash recovery solution, we reduce the state transmission delay by loading a document state from the local permanent storage instead of a remote node. If the node is in the recovery status, the node rejoins the system by loading the local document state and propagating a recovery message r. Then, the node waits for replies from other peer nodes. Algorithm 2 outlines the procedure for a node with transient failures rejoining the system by loading a local document state.

Without losing generality, we assume that at time θ when node s_i has failed, s_i generates the latest document checkpoint at time σ, followed by the recovery procedure that loads the checkpoint and transmits the recovery message r at time γ.

It is crucial for the restored node to decide when it can start generating the operations again. In fact, the failed node s can begin operations only if it has received all lost operations generated at s_i between σ and θ from the other nodes. To prove the correctness of this statement, we introduce and prove Theorem 3.6. Before presenting Theorem 3.6, we describe the property of time stamp and Lemma 3.5.

Property 1. Let O be an operation generated at s and time stamped by SV_O. After executing O at node s, state vector $SV_O[s]$ can be derived from $SV[s]$ as $SV_O[s] = SV[s] + 1$, where SV is the current local state vector.

Lemma 3.5. Given two operations O and O' issued at the same node s_i, the ith sector in their time stamp are different. Thus, we have $\forall O, O', 1 \leq i \leq n$: $s(O) = s(O') = i, O \neq O' \rightarrow SV_O[i] \neq SV_{O'}[i]$.

Proof. Because the two operations are issued at the same node (e.g., $s(O) = s(O') = i$), either O is causal order preceding O' or vice versa (i.e., $O \mapsto O'$ or $O' \mapsto O$) (see also Lemma 3.1). Assume $O \mapsto O'$ and that between O and O', s_i generates other $k-1$ ($k > 0$) operations; thus, $O \mapsto O_{k-1} \mapsto \ldots \mapsto O_2 \mapsto O_1 \mapsto O'$, then we have $SV_O[i] = SV_{O_{k-1}}[i] + 1 = SV_{O_{k-2}}[i] + 2 = \ldots = SV_{O'}[i] + k$, where $k > 0$ (Property 1). Hence, we prove that $SV_O[i] \neq SV_{O'}[i]$. We prove Lemma 3.5 in the same manner when $O' \mapsto O$.

Algorithm 2. Let HB_i^t be the history buffer associated with the latest checkpoint that generated at time t. Local operation generation is disabled

```
y_i^{i,t} ← LO(HB_i^t, j):
for 1 ≤ i ≤ n, where i ≠ j do
    y_i^{j,t} ← LO(HB_i^t, j);
    put y_i^{i,t} and y_i^{j,t} into the recovery message;
    send the recovery message to node s_i;
end for
while True do
    waiting for the operations sent from peer nodes;
    if O is the operation which satisfied: SV_O[S(O)] ←
      SV_i[S(O)]+1 and SV_O[k] ≤ SV_i[k], ∀ k ∈ [1, n]; then
    if (S(O) = i and ∀ O' ∈ HB_i: SV_O' ≠ SV_O) or S(O) ≠ i then
    use Undo/Transform-Do/Transform-Redo
    [25] scheme to execute O;
        end if
    else
        O is delayed until two conditions are satisfied;
    end if
    if all missed operations generated at s_i has been
        executed at s_i again then
        Local operation generation is enabled;
    end if
end while
```

Theorem 3.6. Let σ, θ, and γ be the latest checkpoint time, rash time, and crash recovery time at node s_i. s_i can only issue operations after time t ($t > \gamma$), when all operations generated at $s_i$between $\sigma<gt_i(O)<\theta$ execute at node $s_i$again; that is, $\forall$ O: $\sigma < gt_i(O) < \theta \rightarrow e_i(O) \in HB_i^t$.

Proof. We prove this theorem by contradiction. Let us assume that Theorem 3.6 is incorrect, then node s_i generates an operation O_s at time t'>γ, when at least one operation generated at s_i between $\sigma<gt_i(O)<\theta$ does not execute at node s_i again. Thus, we have $\exists$ O: $\sigma<gt_i(O)<\theta \rightarrow e_i(O) \notin HB_i^{t'}$. Let $O_1 \mapsto O_2 \mapsto ... \mapsto O_k$ be k (k> 0) operations generated at s_i between $\sigma<gt_i(O)<\theta$, so we prove that $\forall$ $1 \leq j \leq h$: $e_i(O) \in HB_i^{t'}$ and $\forall$ h+1 $\leq j \leq k$, $e_i(O) \notin HB_i^{t'}$. Assume that when s_i generates the latest checkpoint at time σ, the local state vector is $SV[i] = d$, then after executing O_h on s_i again, $SV[i]$ becomes $d+h$. So, the operation $SV_{O_s}[i] = d + h + 1$. The timestamp of the operation O_{h+1} that has not executed at s_i again is: $SV_{O_{h+1}}[i] = d + h + 1$. Hence, we prove that $SV_{O_s}[i] = SV_{O_{h+1}}[i]$. Because $O_s \neq O_{h+1}$, we have $SV_{O_s}[i] \neq SV_{O_{h+1}}[i]$(see Lemma 3.5), which is a contradiction. This concludes the proof of theorem 3.6.

If after time σ, there is at least one operation from another node that is executed at s_i or s_i generates at least one operation, then the saved local state is inconsistent with the remote state at the other nodes. We articulate this feature in Theorem 3.7. Before the proof of Theorem 3.7, we address five properties pertinent to history buffer as follows.

Property 2. If the generation time of O at node $s_i$is earlier than time t, then $e_i(O)$ is in history buffer HB_i^t; thus, we have $\forall$ O, $1 \leq i \leq n$: $gt_i(O) < t \rightarrow e_i(O) \in HB_i^t$.

Property 3. If the execution time of O ($s(O) \neq i$) at node s_i is earlier than time t, then $e_i(O)$ is in history buffer HB_i^t. Formally, we have $\forall$ O, $1 \leq i \leq n$: $at_i(O) < t \rightarrow e_i(O) \in HB_i^t$.

Property 4. If the generation time of O at s_i is later than time t, then $e_i(O)$ is not in history buffer HB_i^t. Thus, we formally describe this statement as $\forall$ O, $1 \leq i \leq n$: $gt_i(O) > t \rightarrow e_i(O) \notin HB_i^t$.

Property 5. If the execution time of O ($s(O) \neq i$) at s_i is later than time t, then $e_i(O)$ is not in history buffer HB_i^t. More formally, we have $\forall$ O, $1 \leq i \leq n$: $at_i(O) > t \rightarrow e_i(O) \notin HB_i^t$.

Property 6. Let θ be the time when s_i fails, σ be the time when node s_i generates the latest checkpoint, and γ be the time when s_i begins its crash recovery procedure. For node s_i, history buffer at time γ is the same as that at time δ. We formally describe this statement as $HB_i^{\gamma} = HB_i^{\sigma}$.

Let us assume that $y_i^{j,\sigma} = e_i(O_k)$. We observe that operations O generated at node s_j, ($1 \leq j \leq n$; $j \neq i$), where $gt_j(O_k) < gt_j(O) < at_j(r)$, are also missing in history buffer HB_i^{γ}. The purpose of the crash recovery algorithm is to figure out all the lost operations in node s_i and the effect of their executions is remained unchanged. Hence, we introduce the consistency of the crash recovery as the definition below.

Definition 9. Let σ, θ, and γ be the latest checkpoint time, crash time, and recovery time at node s_i, the crash recovery is consistent iff,

$$\exists t>\gamma,\ \forall O: \\ (\sigma<gt_i(O)<\theta \vee gt_j(O_k)<gt_j(O)<at_j(r)) \\ \rightarrow e_i(O) \in HB_i^t,\ \text{where}\ y_i^{j,\sigma} = e_i(O_k) \quad (1)$$

We devise the GORT algorithm (see Algorithm 3) to obtain the original form of an operation in history buffer.

Let s_j be a node that receives recovery message r from node s_i, ($i \neq j$), s_j responds to the message r at time t. The pseudo code of the GORT algorithm is described below.

Algorithm 3. The Generic Operation Revise Transform algorithm (GORT)

1. Given the history buffer of s_i at time t, HB_i^t = [$e_i(O_1)$, $e_i(O_2)$,..., $e_i(O_k)$], and an operation $e_i(O_j)$ in HB_i^t, the original form of O_j is obtained as follows,
2. Scan HB_i^t from left to right to find the oldest operation $HB_i^t[a]$ that is independent to $e_i(O_j)$;
3. **if** no such operation is found **then**
4.   **return** $O_j \leftarrow e_i(O_j)$;
5. **end if**
6. Scan $HB_i^t[a, j\text{-}1]$ to find all operations that are causally preceding $e_i(O_j)$.
7. **if** no such operation is found **then**
8.   **return** $O_j \leftarrow LET(e_i(O_j)), HB_j^t[a, j\text{-}1]^{-1})$;
9. **end if**
10. $EO_{b_1}^t \leftarrow LET(EO_{b_1}, HB_i^t[a, b_1\text{-}1]^{-1})$;
11. **for** $2 \leq i \leq r$ **do**
12. $TO \leftarrow LET(EO_{b_1}, HB_i^t[a, b_{i-1}]^{-1})$;
13. $EO_{b_1}^t \leftarrow IT(TO, [EO_{b_1}^t, EO_{b_2}^t, ..., EO_{b_{i-1}}^t])$;
14. **end for**
15. $TO \leftarrow LET(EO_{b_1}, HB_i^t[a, j\text{-}1]^{-1})$;
16. **return** $O_j \leftarrow IT(TO, EOL')$;

Let HB_i^t be the history buffer of node s_i at time t, HB_i^t = [$e_i(O_1)$, $e_i(O_2)$,..., $e_i(O_m)$], and $e_i(O_j)$ is an operation in HB_i^t.

In case that $\forall$ $1 \leq k \leq j$-1, $e_i(O_k) \mapsto e_i(O_j)$, then the original form of O_j is the same as its execution form.

Thus, we have $O_j = e_i(O_j)$.

Let $e_i(O_a)$ be the oldest operation that is independent of $e_i(O_j)$. In the simple case that $\forall\ 1 \le k \le a\text{-}1$, $e_i(O_k) \mapsto e_i(O_j)$, and $\forall\ a \le k \le j\text{-}1$, $e_i(O_a) \parallel e_i(O_j)$, then we can directly obtain O_j by applying the list of exclusion transformation function (LET) [25]. Therefore, we obtain $O_j = \text{LET}(e_i(O_j),\ HB_i^t\ [a, j\text{-}1]^{-1})$.

The complicated case is that there is a mixture of independent and dependent operations in the range of $HB_i^t\ [a, j\text{-}1]$. Let EOL = $[EO_{b_1}, EO_{b_2}, ..., EO_{b_r}]$ be the list of operations in the range of $HB_i^t\ [a+1, j\text{-}1]$, which are causally preceding $e_i(O_j)$. EOL' = $[EO'_{b_1}, EO'_{b_2}, ..., EO'_{b_r}]$, EO'_{b_i} is the original form of operation EO'_{b_i}.

For the first operation in list EOL, EO'_{b_1} is derived as $EO'_{b_1} = \text{LET}(EO_{b_1},\ HB_i^t\ [a, b_1\text{-}1]^{-1})$.

For the second operation in list EOL, O_{b_1} is determined by two steps as follows, in which IT is the inclusion transformation function. The detailed information on IT is proposed in [25].

- $TO = LET(EO_{b_2},\ HB_i^t\ [a, b_2\text{-}1]^{-1});$
- $EO'_{b_2} = IT(TO, EO'_{b_1}).$

For the ith operation in list EOL, ($2 \le i \le r$), the following two steps are applied to obtain the corresponding form of operation in EOL.

- $TO = LET(EO'_{b_i},\ HB_i^t\ [a, b_i\text{-}1]^{-1});$
- $EO'_{b_1} = IT(TO, [EO'_{b_1}, EO'_{b_2}, ..., EO'_{b_{i-1}}]).$

If the operation list EOL' is obtained, O_j can be easily obtained by applying the following two steps.

- $TO = LET(EO_{b_1},\ HB_i^t\ [a, j\text{-}1]^{-1});$
- $O_j = IT(TO, EOL').$

After each node s_j executes Algorithm 4, all the lost operations in node s_i will be executed again at node s_i, and the effect of their execution is remained unchanged. Theorem 3.8 below proves the correctness of this statement.

Assumption 1. There is at least one node s_j that, before time $at_j(r)$, has executed all operations generated at the failed s_i between time σ and θ, thus, $\exists\ 1 \le j \le n, j \ne i$, $t<at_j(r)$: $\forall\ O$: $\sigma < gt_i(O) < \theta \to e_j(O) \in HB_i^t$.

Assumption 1 is very essential for the following reason. If no node executes all the lost operations when a recovery message arrives, then some lost operations will never be executed at node s_i again. Consequently, the consistency of the crash recovery cannot be guaranteed.

Theorem 3.8. Our crash recovery algorithm offers a consistent crash recovery.

Proof. Let us assume that $y_i^{i,\sigma} = e_i(O_k)$. For node s_j($1 \le j \le n$, and $j \ne i$), $y_j^{j,\delta} = e_j(LO_j)$ is the latest operation, where $\delta = at_j(r)$ is the arrival time of recovery message r from s_i to s_j. At time $t_j = at_i(LO_j)$, $e_i(LO_j)$ is residing in history buffer HB_i^{tj} (i.e., $e_i(LO_j) \in HB_i^{tj}$) (see also Definition 5). Since the crash recovery algorithm re-sends operations, which satisfy $s(O) = j$ and $O \mapsto y_i^{j,\delta}$, to s_i; $y_i^{j,\sigma} \mapsto y_j^{j,\delta}$; hence, $y_j^{j,\delta}$ is sent to s_i again. Because $\forall\ e_j(O) \in HB_j^{\delta}$: $s(O) = j \to (O \mapsto y_j^{j,\delta})$ (see Definition 8), we prove that at time t_j, $\forall$ O: $gt_j(O_k)<gt_j(O)<at_j(r) \to e_i(O) \in HB_i^{tj}$ (see the property of causality preservation). Thus, we obtain

$$t_\alpha = \max_{1 \le j \le n, j \ne i}(t_j) = \max_{1 \le j \le n, j \ne i}(at_i(LO_j)) \quad \textbf{(2)}$$

At time t_α, we have $\forall\ O$, $1 \le j \le n$, $j \ne i$: $gt_j(O_k)<gt_j(O)<at_j(r) \to e_i(O) \in HB_i^{t\alpha}$. (1)

Algorithm 4. The algorithm in s_j to respond to the message r. Get $O_a \leftarrow y_i^{i,\sigma}$ and $O_b \leftarrow y_i^{j,\sigma}$ from the recovery message.

```
k ← 1
b_i ← false;
b_j ← false;
if y_i^{i,σ} = φ then
b_i ← true;
end if
if y_i^{j,σ} = φ then
    b_j ← true;
end if
while k ≤ |HB_j^t| do
O ← HB_j^t [k];
    if b_i = false then
   if SV_O = SV_{O_a} then
b_i ← true;
end if
    else
    if S(O) = i then
          send O' ← GORT(O) to s_i;
       end if
     end if
if b_j = false then
if SV_O = SV_{O_b} then
          b_j ← true;
       end if
     else
       if S(O) = j then
          send O' ← GORT(O) to s_i;
       end if
     end if
     k ← k + 1;
end while
```

According to assumption 1, let s_k be the node that has executed all operations issued at node s_i between σ and θ; thus, we have $\exists\ t<\delta$: $\forall\ O$: $\sigma<gt_i(O)<\theta \to e_k(O) \in HB_k^t$. Therefore, we obtain $\forall\ O$: $\sigma<gt_i(O)<\theta \to$

$e_k(O)\ HB_k^{\delta} \in$ (2).

Let δ be the arrival time of the crash recovery message from s_i to s_k, $\delta = at_k(r)$, and $y_k^{i,\delta} = e_k(LO_k')$ is the latest operation from s_i in HB_k^{δ}. As described in our algorithm, these operations are delivered back to node s_i again; we then obtain $\exists\ t_\beta = at_i(LO_k') > \delta$: $e_i(LO_k') \in HB_i^{t\beta}$. Because $\forall\ e_k(O) \in HB_k^{\delta}$, $s(O) = i \rightarrow (O \Mapsto LO_k')$, we prove that at time t_β, it is true that $\forall\ e_k(O) \in HB_k^{\delta}$: $s(O) = i \rightarrow e_i(O) \in HB_i^{t\beta}$ (3) (see the causality property).

Based on items (2) and (3) above, we prove that at time t_β, $\forall\ O$: $\sigma < gt_i(O) < \theta \rightarrow e_i(O) \in HB_i^{t\beta}$ (4). According to items (1) and (4), we have $\exists t = \max(t_\alpha, t_\beta) > \gamma$: $\forall\ O$: $(s(O) = i \wedge \sigma < gt_i(O) < \theta) \vee (s(O) = j \neq i \wedge gt_j(O_k) < gt_j(O) < at_j(r)) \rightarrow e_i(O) \in HB_i^{t}$, where $y_i^{j,\delta} = e_i(O_k)$. Thus, the crash recovery is consistent, which concludes the proof of the theorem.

4 Performance Analysis

Now we are in a position to evaluate the performance of our new approach of recovery support for distributed editing systems. We assume that when a node leaves the distributed editing system successfully, it has created m document checkpoints. The expected interval between the time a node joins and leaves the system reflects the performance of the editing system.

$P_i(2 \leq i \leq m)$ in Figure 2 represents the execution time on a node, it is the nominal measured in CPU cycles between (i-1)th and ith checkpoints. P_1 indicates the interval between the beginning of the node and its first checkpoint without any transient failure. The total execution time is measured as $P = \sum_{i=1}^{m} P_i$.

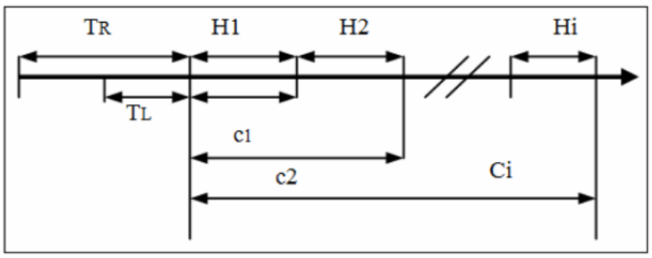

Figure 2. Definition for c_i, H_i, T_L, and T_R

Let $c_i(1 \leq i \leq m)$ be the execution time from the beginning of a node to the ith checkpoint in presence of the node or link failures. Let C_i denote the expected value of c_i, $C_i = E(c_i)$. Thus, the expected interval between the time a node joins and leaves the distributed editing system is $C_m = E(c_m)$.

Transient failures of a node and a network link can be recovered by either loading local document states or remote document states. Let p and q be the probability of recovering a node by using our new LDS approach and the traditional RDS approach, respectively; it is clear that $p+q = 1$. Let T_L and T_R denote time overhead for retrieving local document states and remote document states, respectively. $f_i(t)(i \in [2, m])$ denotes the probability of a node/link failure in t units of time from the time of the (i-1)th checkpoint. $f_1(t)$ is the failure probability from the very beginning. Then, we have

$$C_1 = \begin{cases} P_1 & \text{with probability } 1 - f_i(P_i) \\ P_1 + T_L + C_1 & \text{with probability } p \times f_i(P_i) \\ P_1 + T_R + C_1 & \text{with probability } q \times f_i(P_i) \end{cases} \quad \textbf{(3)}$$

Let H_i represent the time interval between (i-1)th and ith checkpoint. Thus, we have

$$C_i = C_{i-1} + H_i + T_C \quad \textbf{(4)}$$

$$H_1 = c_1 \quad \textbf{(5)}$$

$$H_i = \begin{cases} P_i & \text{with probability } 1 - f_i(P_i) \\ P_i + T_L + C_i & \text{with probability } p \times f_i(P_i) \\ P_i + T_R + C_i & \text{with probability } q \times f_i(P_i) \end{cases} \quad \textbf{(6)}$$

where $2 \leq i \leq m$.

C_i is derived from Equation 6 as the equation below, where $2 \leq i \leq m$,

$$C_i = \frac{C_{i-1} + P_i + (pT_L + qT_R) f_i(P_i)}{1 - f_i(P_i)} \quad \textbf{(7)}$$

C_m represents the expected interval between the time the node joins and leaves the system; C_m is obtained by repeatedly applying the above equation m-1 times,

$$C_m = \sum_{j=1}^{m} \prod_{i=j}^{m} \frac{P_j + (pT_L + qT_R) f_j(P_j)}{1 - f_i(P_i)} \quad \textbf{(8)}$$

The value of C_m represents the performance of the evaluated distributed editing system. Hence, in order to optimize the performance, one can minimize C_m by determining the proper checkpointing frequency. The value of m that minimizes the equation 8 is an optimal one.

Let $C^L(P, k)$ denote the execution time of the node in the presence of up to k recovering by loading a local document state, let p_i^S and p_i^U be the probability of the ith LDS approach becoming successful and unsuccessful, respectively, where $p_i^S + p_i^U = 1$. $C^L(P, k)$ is given as below,

$$\begin{aligned} C^L(P, k) &= \\ &(P+T_L)\, p_1^S + 2(P+T_L)\, p_1^U p_2^U + \ldots + k(P+T_L) \\ &\prod_{i=1}^{k-1} p_i^U p_k^S + [k(P+T_L) + \frac{p + T_R}{1 - f_1(p)}] \prod_{i=1}^{k} p_i^U \quad \textbf{(9)} \\ &= \sum_{j=1}^{k-1} [j + (P + T_L)] \prod_{i=1}^{j-1} p_i^U p_j^S + k(P+T_L) \end{aligned}$$

$$\prod_{i=1}^{k-1} p_i^U + [\frac{p+T_R}{1-f_i(p)}]\prod_{i=1}^{k-1} p_i^U$$

The values of p_i^S and p_i^U are not known until the (*i*-1)th unsuccessful LDS recovery occurs. We derive the approximate probability for p_i^S and p_i^U. With an increased number of unsuccessful crash recoveries, the probability of permanent rises. Thus,

$$p_1^U < p_2^U < \cdots < p_k^U \text{ and } p_1^S < p_2^S < \cdots < p_k^S \quad \textbf{(10)}$$

We assume that $\frac{p_i^S}{p_{i-1}^S}$ = w_i< 1, and for the simplicity, it is assumed that $w_1 = w_2 = ... = w_k = w$, and $p_1^S = p$. Equation 11 is derived from Equation 10 as follows.

$$CL(P, k) = \sum_{j=1}^{k-1}[j(P+T_L)]\prod_{i=1}^{j-1}(1-pw^{i-1}) + k(P+T_L) \prod_{i=1}^{k-1}(1-pw^{i-1}) + k(P+T_L) \prod_{i=1}^{k-1}(1-pw^{i-1}) \quad \textbf{(11)}$$

The time overhead of LDS recovery is determined by *P* and the arrival rate of operations *λ*. Suppose the operation arrival rate is constant, hence, with the increase of *P*, the probability of successful LDS recovery decreases, and the time overhead of the unsuccessful LDS also increases. On the other hand, the time overhead of RDS recovery is decided by the data volume associated with the context of the document. For the simplicity, we assume that the cost of the RDS recovery remains constant, and it is modelled as follows,

$$C^R(R) = \frac{P+T_R}{1-f_1(P)} \quad \textbf{(12)}$$

LDS crash recovery is an efficient method to recover the temporary failures in node and links. It continues working until the permanent failure occurs (checkpoint stored on local storage is missing) or the time overhead of LDS recovery is larger than RDS recovery. Thus, given value *P*, $C^L(P, k)$ can be determined by *k*, which must satisfy $C^L(P, k)<C^R(P)$.

Table 1 describes the relation between *k* and $C^L(P, k)$. *P* is set to 100, 200, and 300, respectively. C^L first decreases with the increase of *k*, and when *k* = 12, C^L is then minimized. After *k* = 12, C^L rises with the increase of *k*. In this case, 12 is the optimal value for *k*.

Table 1. T_L=20, T_R=40, *w*=0.8, *p*=0.8, $f_1(P)$=0.1

k	2	4	6	8	10	12
P=100	179.2	177.6	170.5	167.7	166.7	166.1
P=200	327.2	325.2	312.3	307.3	305.5	305.0
P=300	475.2	473.9	454.2	447.0	444.3	443.6
k	14	16	18	30	50	100
P=100	166.5	166.9	167.4	171.6	179.6	199.8
P=200	305.2	305.9	306.8	314.6	329.1	366.2
P=300	443.9	444.9	446.2	457.4	478.7	532.6

To evaluate the impact of the probability of the first successful LDS recovery on $C^L(P, k)$, we fix T_L, T_R, *w*, and $f_1(P)$, and increased *k* from 10 to 30 with an increment of10. Table 2 shows the execution time of the node in thepresence of up to *k* LDS recovery as a function of *p*. The higher the probability *p* is, the less execution time of the node in the presence of up to *k* LDS recovery is. It suggests that a higher probability of the first successful LDS recovery results in a better performance.

Table 2. *P* = 100, T_L = 20, T_R = 40, *w* = 0.8, $f_1(P)$ = 0.1

p	0.65	0.70	0.75	0.80	0.85	0.90
k=10	208.5	192.9	179.2	166.7	154.8	143.3
k=20	217.4	197.9	181.8	168.0	155.4	143.4
k=30	234.3	208.5	188.2	171.6	157.3	144.3

Table 3 illustrates the relation between *w* and $C^L(P, k)$. T_L, T_R, *p*, and $f_1(P)$ are fixed, and *k* is set to 10, 20, and 30, respectively. Like the effect of *p* on C^L, as the value of w rises, the execution time of the failed node in the presence of up to *k* LDS recovery decreases. This is because with the increase of value w, the probability of ith unsuccessful LDS recovery decreases, and as p_i^U drops, C^L decreases. This suggests that if we could increase the probability of the successful LDS recovery, the performance of the system would be enhanced.

Table 3. *P* = 100, T_L = 20, T_R = 40, *w* = 0.8, $f_1(P)$ = 0.1

w	0.65	0.70	0.75	0.80	0.85	0.90
k=10	195.1	181.9	172.5	166.7	164.9	166.0
k=20	232.0	202.7	180.7	168.0	164.0	165.5
k=30	270.7	226.1	192.1	171.6	164.4	165.4

5 Related Work

Distributed editing systems have been studied deeply [4, 8, 12, 18, 26]. Real-time distributed editing systems are most effective during the initial and integration/reviewing stages of distributed authoring [6, 23]. On the other hand, non-real-time distributed systems work efficiently for cooperation in authoring team. Table 4 displays a comparison between these real-time and non-real-time systems.

5.1 Non-real-time Systems

Non-real-time distributed editing systems have shared documents that can be accessed and locked separately. A shared repository, such as distributed file system, serves as the infrastructure for many non-real-time distributed systems [5, 13-14]. WebDAV is an application-layer network protocol offering capabilities to support remote collaborative authoring, metadata management, version control, and configuration management [5]. Unique operations implemented in

Table 4. Method comparison

	Whitehead and Goland [5]	PREP [13]	Pacull et al. [14]	Koch [9]	Sun et al. [25]	Yang et al. [27]	Beck and Bellotti [2]	Shim and Prakash [19]	Our method
Non-real-time	✓	✓	✓						
Real-time				✓	✓	✓	✓	✓	✓
Fault tolerance				✓				✓	✓
Consistency maintenance					✓				✓
Fail recovery		✓							✓

WebDAV include overwrite prevention, properties, and namespace management.

The *flexible diff* system reports differences among multiple text versions. This system provides flexible control operations, allowing users to configure reported changes [13]. Our editing system is distinct from the aforementioned systems in the way that ours facilitates collaborative authoring in a real-time manner.

5.2 Real-time Systems

Most existing studies in real-time distributed editing systems focus on user intention preservation [10], consistency maintenance [2, 21, 25, 27], group undo [22], and group awareness [7, 15, 28]. Fault tolerance and crash recovery issues, however, have not been studied extensively. If a real-time distributed editing system is to be efficiently used over a wide area network, the fault-tolerant issues must be taken into account, for the reason that wide area networks are usually unreliable [19]. If group communication subsystems are designed and implemented properly, they can provide an infrastructure for building distributed and reliable services on top of their message broadcasting and membership services [1] [11]. The drawback of these systems is that they do not directly manage group-shared application state and transfer group state to new nodes.

Koch [9] studied the requirements for distributed editing systems; Koch also proposed a model, in which fault tolerance is introduced. This technique is also discussed in [1]. Zhao et al. [30] investigated Byzantine fault tolerance for collaborative editing systems with commutative operations. But they do not consider the consistency maintenance, which is fully taken into account in our approach. PREP [13] is a distributed writing system that uses the concept of flexible diffing for reporting differences between versions of texts. But our algorithm is devised for real-time distributed editing systems. Nicolaescu et al. [29] studied multiple communication protocols, and developed a near real-time lightweight framework for collaborative editing of arbitrary data types in peer-to-peer settings. But we investigate the real-time distributed editing systems in a general distributed environment.

6 Conclusion and Future Work

We address the crash recovery issues in the context of real-time distributed systems. An efficient recovery algorithm is presented to make the real-time distributed systems more reliable. In our new approach, each node maintains a local document state, which is generated periodically. If a failure occurs in the node or links, the node is able to rejoin the distributed editing systems.

We studied the factors that affect this interval time and derived an equation to determine such interval time, and the performance of the system can be optimized by determining a proper frequency of generating a document state.

In future, we will extend this work by devising garbage collection techniques for reclaiming the history buffer.

Acknowledgements

The authors want to thank Mojen Lau for proofreading the final presentation of this paper. Xiao Qin's work was supported by the US National Science Foundation under Grants CCF-0845257(CA-REER), CNS-0757778 (CSR), CCF-0742187 (CPA), CNS-0917137 (CSR), CNS-0831502 (CyberTrust), CNS-0855251 (CRI), OCI-0753305 (CI-TEAM), DUE-0837341 (CCLI), and DUE-0830831 (SFS). Mohammed I. Alghamdi's work was supported by Al-Baha University.

References

[1] Y. Amir, D. Dolev, S. Kramer, D. Malki, *Transis: A Communication Sub-System for High Availability*, Technical Report TR CS91-13, April, 1992.

[2] E. E. Beck, V. M. E. Bellotti, Informed Opportunism as Strategy: Supporting Coordination in Distributed Collaborative Writing, *Proceedings of the Third European Conference on Computer-Supported Cooperative Work, Milan*, Italy, 1993, pp. 233-248.

[3] D. Chen and C. Sun, A Distributed Algorithm for Graphic Objects Replication in Real-time Group Editors, *Proceedings of the International ACM SIGGROUP Conference on Supporting Group Work*, Phoenix, AZ, 1999, pp. 121-130.

[4] A. Craig, A. Davoust, B. Esfandiari, A Distributed Wiki System based on Peer-to-peer File Sharing Principles, *2011 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology (WI-IAT)*, Lyon, France, 2011, pp. 364-371.

[5] Jr. E. J. Whitehead, Y. Y. Goland, WebDAV: A Network Protocol for Remote Collaborative Authoring on the Web, *Proceedings of the Sixth European Conference on Computer-Supported Cooperative Work*, Copenhagen, Denmark, 1999, pp. 291-310.

[6] H. Fan, C. Sun, Dependency-based Automatic Locking for Semantic Conflict Prevention in Real-time Collaborative Programming, *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, Trento, Italy, 2012, pp. 737-742.

[7] H. Higaki, K. Tanaka, M. Takizawa, Protocol for Pseudo-active Replication in Wide-area Networks, *Proceedings of the Tenth International Workshop on Database and Expert Systems Applications*, Florence, Italy, 1999, pp. 678-682.

[8] T. Kärkkäinen, J. Ott, Shared Content Editing in Opportunistic Networks, *Proceedings of the 9th ACM MobiCom Workshop on Challenged Networks*, Maui, HI, 2014, pp. 61-64.

[9] M. Koch, Design Issues and Model for a Distributed Multi-user Editor, *Computer Supported Cooperative Work (CSCW)*, Vol. 3, No. 3-4, pp. 359-378, September, 1994.

[10] D. Li, L. Zhou, R. R. Muntz, A New Paradigm of User Intention Preservation in Realtime Collaborative Editing Systems, *Proceedings of the Seventh International Conference on Parallel and Distributed Systems*, Iwate, Japan, 2000, pp. 401-408.

[11] S. Mishra, L. L. Peterson, R. D. Schlichting, Consul: A Communication Substrate for Fault-Tolerant Distributed Programs, *Distributed Systems Engineering*, Vol. 1, No. 2, pp. 87-103, December, 1993.

[12] B. Nédelec, P. Molli, A. Mostefaoui, E. Desmontils, Lseq: An Adaptive Structure for Sequences in Distributed Collaborative Editing, *Proceedings of the 2013 ACM Symposium on Document Engineering*, Florence, Italy, 2013, pp. 37-46.

[13] C. M. Neuwirth, R. Chandhok, D. S. Kaufer, P. Erion, J. Morris, D. Miller, Flexible Diff-ing in a Collaborative Writing System, *Proceedings of the the 1992 ACM Conference on Computer-supported Cooperative Work*, Toronto, Canada, 1992, pp. 147-154.

[14] F. Pacull, A. Sandoz, A. Schiper, Duplex: A Distributed Collaborative Editing Environment in Large Scale, *Proceedings of the 1994 ACM Conference on Computer Supported Cooperative Work*, Chapel Hill, North Carolina, 1994, pp. 165-173.

[15] W. Prinz, NESSIE: An Awareness Environment for Cooperative Settings, *Proceedings of the Sixth European Conference on Computer Supported Cooperative Work*, Copenhagen, Denmark, 1999, pp. 391-410.

[16] X. Qin, Z. Han, H. Jin, L. Pang, S. Li, Real-time Fault-tolerant Scheduling in Heterogeneous Distributed Systems, *Proceedings of the 2000 International Conference on Parallel and Distributed Processing Techniques and Applications*, Las Vegas, NV, 2000, pp. 26-29.

[17] M. Ressel, D. Nitsche-Ruhland, R. Gunzenhäuser, An Integrating, Transformation-oriented Approach to Concurrency Control and Undo in Group Editors, *Proceedings of the 1996 ACM Conference on Computer Supported Cooperative Work*, Boston, Massachusetts, 1996, pp. 288-297.

[18] M. A. Sasse, M. J. Handley, Collaborative Writing With Synchronous and Asynchronous Support Environments, in: R. Rada (Ed.), *Groupware and Authoring*, Academic Press, 1996, pp. 205-218.

[19] H. S. Shim, A. Prakash, Tolerating Client and Communication Failures in Distributed Groupware Systems, *Proceeding of the 17th IEEE Symposium on Reliable Distributed Systems*, West Lafayette, IN, 1998, pp. 221-227.

[20] K.-F. Ssu, B. Yao, W. K. Fuchs, An Adaptive Checkpointing Protocol to Bound Recovery Time with Message Logging, *Proceeding of the 18th IEEE Symposium on Reliable Distributed Systems*, Lausanne, Switzerland, 1999, pp. 244-252.

[21] C. Sun, Y. Yang, Y. Zhang, D. Chen, Distributed Concurrency Control in Real-time Cooperative Editing Systems, *Proceedings of the Second Asian Computing Science Conference on Concurrency and Parallelism, Programming, Networking, and Security*, Singapore, 1996, pp. 84-95.

[22] C. Sun, Undo Any Operation at Any Time in Group Editors, *Proceedings of the 2000 ACM Conference on Computer Supported Cooperative Work, CSCW '00*, Philadelphia, PA, 2000, pp. 191-200.

[23] C. Sun, D. Chen, Consistency Maintenance in Real-time Collaborative Graphics Editing Systems, *ACM Transactions on Computer-Human Interaction (TOCHI)*, Vol. 9, No. 1, pp. 1-41, March, 2002.

[24] C. Sun, C. Ellis, Operational Transformation in Real-time Group Editors: Issues, Algorithms, and Achievements, *Proceedings of the 1998 ACM Conference on Computer Supported Cooperative Work*, Seattle, WA, 1998, pp. 59-68.

[25] C. Sun, X. Jia, Y. Zhang, Y. Yang, D. Chen, Achieving Convergence, Causality Preservation, and Intention Preservation in Real-time Cooperative Editing Systems, *ACM Transactions on Computer-Human Interaction*, Vol. 5, No. 1, pp. 63-108, March, 1998.

[26] D. Sun, C. Sun, Context-based Operational Transformation in Distributed Collaborative Editing Systems, *IEEE Transactions on Parallel and Distributed Systems*, Vol. 20, No. 10, pp. 1454-1470, October, 2009.

[27] Y. Yang, C. Sun, Y. Zhang, X. Jia, Real-Time Cooperative Editing on the Internet, *IEEE Internet Computing*, Vol. 4, No. 3, pp. 18-25, May/June, 2000.

[28] Y. Yokota, H. Tarumi, and Y. Kambayashi, Extended Awareness Support for Cooperative Work in Non-WYSIWIS Condition, *Proceedings of 5th International Computer Science Conference*, Hong Kong, China, 1999, pp. 186-195.

[29] P. Nicolaescu, K. Jahns, M. Derntl, R. Klamma, Yjs: A Framework for Near Real-Time P2P Shared Editing on Arbitrary Data Types, *15th International Conference on Web Engineering (ICWE'15)*, Rotterdam, The Netherlands, 2015,

pp. 675-678.
[30] W. Zhao, M. Babi, W. Yang, X. Luo, Y. Zhu, J. Yang, C. Luo, M. Yang, Byzantine Fault Tolerance for Collaborative Editing with Commutative Operations, *IEEE International Conference on Electro Information Technology (EIT)*, Grand Forks, ND, 2016, pp. 246-251.

Biographies

Mohammed I. Alghamdi received the B.S. degree in Computer Science from King Saud University, Riyadh, Saudi Arabia in 1999. He received the M.S. degrees in Computer Science from Colorado Technical University, Denver, Colorado in 2003. He received the Ph.D. degree in Computer Science from New Mexico Institute of Mining and Technology in 2008. Currently, he is an Assistant Professor with the Department of Computer Science, Al-Baha University, Kingdom of Saudi Arabia. His research interests include wireless networks, storage systems, parallel and distributed systems, and computer system security. He is a senior member of IEEE.

Xunfei Jiang is an Assistant Professor in the Department of Computer Science at Earlham College. She received the B.S. and M.S. degrees in Computer Science from Huazhong University of Science and Technology (HUST), China, in 2004 and 2007. She received the Ph.D. degree in the Department of Computer Science and Software Engineering at Auburn University in 2014. Her research interests include parallel and distributed systems, energy-efficient storage systems, thermal modeling, and hybrid data storage systems.

Ji Zhang received his B.S. and M.S. degrees in Computer Science from Huazhong University of Science and Technology, Wuhan, China in 2004 and 2007, respectively. He also obtained the Ph.D. degree in Computer Science from Auburn University in 2013. Currently, he is working as a senior software engineer at Doxpop LLC. His research interests include I/O-intensive computation, parallel and distributed file systems, and geographic information systems.

Jifu Zhang received the BS and MS in Computer Science and Technology from Hefei University of Technology, China, in 1983 and 1989, respectively. He received the Ph.D. degree in Pattern Recognition and Intelligence Systems from Beijing Institute of Technology in 2005. He is currently a Professor in the School of Computer Science and Technology at TYUST. His research interests include data mining and artificial intelligence, parallel and distributed systems.

Xiao Qin received the B.S. and M.S. degrees in Computer Science from the Huazhong University of Science and Technology, Wuhan, China, and the Ph.D. degree in Computer Science from the University of Nebraska-Lincoln, Lincoln, in 1992, 1999, and 2004, respectively. Currently, he is an Associate Professor with the Department of Computer Science and Software Engineering, Auburn University, Auburn, AL. His research interests include parallel and distributed systems, storage systems, fault tolerance, real-time systems, and performance evaluation.