Recovery Support for Real-time Distributed Editing Systems

Mohammed I. Alghamdi¹, Xunfei Jiang², Ji Zhang³, Jifu Zhang⁴, Xiao Qin³

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.

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
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 $S={s_1, s_2, \ldots, s_n}$. Node $s_i$ is a node involved in editing work. $C$ is a finite set of channels, i.e., $C=\{c_{ij}, 1 \leq i \leq n, i\neq j \leq n\}$, 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. $LDS$ 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 $s_i$, $LDS$ 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, $e(O)$ represents the execution form of $O$ at $s_i$, $gt(O)$ denotes the time when $s_i$ generates $O$, and $at(O)$ represents the execution time of $O$ at the remote site $s_j$. It is certain that $gt(O) \rightarrow s(O) = i$, and $at(O) \rightarrow s(O) \neq i$.

The above definition illustrates two implications. First, $gt(O)$ implies that the node of operation $O$ is node $i$ (i.e., $s(O) = i$). Second, $at(O)$ 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 $O_i$ and $O_j$, $O_i$ is causal order preceding $O_j$, denoted by $O_i \Rightarrow O_j$, iff:

1. $s(O_j) = s(O_i) = k$, and $gt(O_i) < gt(O_j)$; or
2. $s(O_j) \neq s(O_i)$, $at(O_i) < gt(O_i)$, where $k = s(O_j)$; or
3. There exists an operation $O_k$, such that $O_i \Rightarrow O_k$, $O_k \Rightarrow O_j$.

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, $O_i$ is created on node $k$ and $O_j$ 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. $gt(O_j)$).

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 $O_j$ 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 $O_i \Rightarrow O_j$, nor $O_j \Rightarrow O_i$, which is defined as $O_i \parallel O_j$.

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 $CT_{O_i} = CT_{O_j} + [O_j]$.

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

In a distributed editing system, each node $s_i$ maintains a status $\xi_i$, 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.

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 $O_j$ is causal order preceding $O_i$ (i.e., $O_j \Rightarrow O_i$), 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 $HB'_j$ 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., $O_i \Rightarrow O_j$) or $O_j$ is causal order preceding $O_i$ (i.e., $O_j \Rightarrow O_i$).

**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., $O_i \Rightarrow O_j$), then $O_j$ is total order preceding $O_i$ (i.e., $O_j \Rightarrow O_i$). Thus, we have $\forall O_i, O_j: (s(O_i) \Rightarrow s(O_j)) \rightarrow (O_j \Rightarrow O_i)$.

**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 preceding to $O_j$ (i.e., $O_i \Rightarrow O_j$), then $O_i$ is causal order preceding $O_j$ (i.e., $O_i \Rightarrow O_j$). Thus, we have $\forall O_i, O_j \in$
HB: \( s(O_j) = s(O_i) \land O_i \Rightarrow O_j \rightarrow (O_i \Rightarrow O_j) \).

**Proof.** We prove this lemma by contradiction. Assuming that lemma 3.3 is incorrect, we show that either \( O_i \) is causal order preceding \( O_j \) (i.e., \( O_i \Rightarrow O_j \)) or \( O_j \) and \( O_i \) 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 \( O_i \) is causal order preceding \( O_j \) (i.e., \( O_i \Rightarrow O_j \)). Thus, \( O_i \) is total order preceding \( O_j \) (i.e., \( O_i \Rightarrow O_j \)) (see lemma 3.2), which is a contradiction. This proves the lemma. □

Given \( HB_i \) 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 \( HB_i \) and node \( s_j \), Algorithm \( LO(HB_i', j) \) determines the latest operation issued at node \( s_j \) y'.

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

**Algorithm 1.** \( LO(HB_i', j) \): Given a history buffer \( HB_i' \) at node \( s_j \), \( y_j' \) is the latest operation generated at \( s_j \) it is obtained as the follows

1. \( j \leftarrow \mid HB_i' \mid \)
2. **while** \( j > 0 \) **do**
3. \( O \leftarrow HB_i'[j] \)
4. **if**\( s(O) = j \)**
5. \( \text{return } y_j' = O \)
6. **else**
7. \( j \leftarrow j - 1 \)
8. **end if**
9. **end while**
10. **return**

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 rejoin 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 \( \theta \) when node \( s_i \) has failed, \( s_i \) generates the latest document checkpoint at time \( \sigma \), followed by the recovery procedure that loads the checkpoint and transmits the recovery message \( r \) at time \( \gamma \).

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 \( s \), between \( \sigma \) and \( \theta \) 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 \( SV[s] = SVo[s] + 1 \), where \( SV \) is the current local state vector.

**Lemma 3.5.** Given two operations \( O \) and \( O' \) issued at the same node \( s \), 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., \( O \Rightarrow O' \) or \( O' \Rightarrow O \)) (see also Lemma 3.1). Assume \( O \Rightarrow O' \) and that between \( O \) and \( O' \), the \( i \)th operation is \( r \). Then, we can prove \( SV_o[i] = SV_o[i] + 1 = SV_o[i] + 2 = ... = SV_o[i] + k \), 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 \( O \Rightarrow O' \).

**Algorithm 2.** Let \( HB_i' \) be the history buffer associated with the latest checkpoint that generated at time \( t \). Local operation generation is disabled

1. \( y_j^{\alpha} \leftarrow LO(HB_i', j) \)
2. **for** \( i \leq i \leq n \), where \( i \neq j \** **do**
3. \( y_j^{\alpha} \leftarrow LO(HB_i', j) \)
4. put \( y_j^{\alpha} \) and \( y_j^{\alpha} \) into the recovery message;
5. send the recovery message to node \( s_i \)
6. **end for**
7. **while**\( True \** **do**
8. waiting for the operations sent from peer nodes;
9. if \( O \) is the operation which satisfied: \( SV_o(S(O)]\leftarrow SV_o(S(O)]+1 \land SV_o[k] = SV_o[k], \forall k \in [1, n] \); **then**
10. if \( S(O) = i \) and \( \forall O' \in HB_i, SV_o[k] \neq SV_o[k] \); **then**
11. use Undo/Transform-Do/Transform-Redo [25] scheme to execute \( O \)
12. **end if**
13. **else**
14. \( O \) is delayed until two conditions are satisfied;
15. **end if**
16. if all missed operations generated at \( s_i \) has been executed at \( s_i \) again **then**
17. Local operation generation is enabled;
18. **end if**
19. **end while**
Theorem 3.6. Let σ, θ, and γ be the latest checkpoint time, rash time, and crash recovery time at node si, sj can only issue operations after time t (t > γ), when all operations generated at sj between σ<gtj(O)<θ execute at node si again; that is, ∀O: σ < gtj(O) < θ → ei(O) ∈ HB′i.

Proof. We prove this theorem by contradiction. Let us assume that Theorem 3.6 is incorrect, then node sj generates an operation Oi at time t > γ, when at least one operation generated at sj between σ<gtj(O)<θ does not execute at node si again. Thus, we have ∃ O: σ<gtj(O)<θ → ei(O) ∉ HB′i. Let Oi⇒Oj⇒...⇒Ok be k (k > 0) operations generated at sj between σ<gtj(O)<θ, so we prove that ∀1 ≤ j≤ h : ei(O) ∈ HB′i and ∀ h+1 ≤ j≤ k, ei(O) ∉ HB′i. Assume that when sj generates the latest checkpoint time at σ, the local state vector is SV[i] = d, then after executing Oi, on sj again, SV[i] becomes d+h. So, the operation SVsj[i] = d + h + 1. The timestamp of the operation Oj(i = h), that has not executed at si again is: SVsj[i] = d + h + 1. Hence, we prove that SVsj[i] ≠ SVsj[i]. Because Oj ≠ Oh+i, we have SVsj[i] ≠ SVsj[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 si or sj 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 si is earlier than time t, then ei(O) is in history buffer HB′i; thus, we have ∀ O, 1 ≤ i ≤ n: gtj(O) < t → ei(O) ∈ HB′i.

Property 3. If the execution time of O (s(O) ≠ i) at node si is earlier than time t, then ei(O) is in history buffer HB′i. Formally, we have ∀ O, 1 ≤ i ≤ n: atj(O) < t → ei(O) ∈ HB′i.

Property 4. If the generation time of O at si is later than time t, then ei(O) is not in history buffer HB′i. Thus, we formally describe this statement as ∀ O, 1 ≤ i ≤ n: gtj(O) > t → ei(O) ∉ HB′i.

Property 5. If the execution time of O (s(O) ≠ i) at si is later than time t, then ei(O) is not in history buffer HB′i. More formally, we have ∀ O, 1 ≤ i ≤ n: atj(O) > t → ei(O) ∉ HB′i.

Property 6. Let θ be the time when sj fails, σ be the time when node sj generates the latest checkpoint, and γ be the time when sj begins its crash recovery procedure. For node sj, history buffer at time γ is the same as that at time δ. We formally describe this statement as HB′i = HB′i.

Let us assume that yj|σ = ei(Oj). We observe that operations O generated at node sj (1 ≤ j ≤ n; j ≠ i), where gtj(Oj) < gtj(O) < atj(r), are also missing in history buffer HB′i. The purpose of the crash recovery algorithm is to figure out all the lost operations in node sj 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 sj, the crash recovery is consistent iff,

\[ \exists\sigma, \theta, \gamma\quad \forall O: \quad (\sigma<gtj(O)<\theta \vee gtj(O)<\theta<at(r)) \rightarrow e(O) \in HB′i, \text{ where } y_j|\sigma = e(Oj) \]

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

Let si be a node that receives recovery message r from node sj (i ≠ j), si 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 si at time t, HB′i = [ei(Oj), e2(Oj),... , en(Oj)], and an operation e(Oj) in HB′i, the original form of Oj is obtained as follows.
2. Scan HB′i from left to right to find the oldest operation HB′i[a] that is independent to e(Oj);
3. if no such operation is found then
4. return Oj ← e(Oj);
5. end if
6. Scan HB′i[a] to find all operations that are causally preceding e(Oj).
7. if no such operation is found then
8. return Oj ← LET(e(Oj)), HB′i[a, j−1]−1;
9. end if
10. E0n′ ← LET(E0n′, HB′i[a, b−1]−1);
11. for 2 ≤ i ≤ n do
12. TO ← LET(E0n′, HB′i[a, b−1]−1);
13. E0λ ← IT(TO, [E0λ, E0λ, ..., E0λ]);
14. end for
15. TO ← LET(E0n′, HB′i[a, j−1]−1);
16. returnOλ ← IT(TO, E0λ);

Let HB′i be the history buffer of node si at time t, HB′i = [e(Oj), e2(Oj),... , en(Oj)], and e(Oj) is an operation in HB′i. In case that ∀ 1 ≤ k ≤ j−1, e(Oj) ⊃ e(Oj), then the original form of Oj is the same as its execution form.
Thus, we have $O_i = e_i(O)$. 

Let $e_i(O)$ be the oldest operation that is independent of $e_j(O)$. In the simple case that $\forall 1 \leq k \leq \alpha - 1, e_i(O) \Rightarrow e_j(O)$, and $\forall s \leq k < j - 1, e_i(O) \parallel e_j(O)$, then we can directly obtain $O_i$ by applying the list of exclusion transformation function (LET) [25]. Therefore, we obtain $O_i = \text{LET}(e_i(O), HB_j [a, j-1])$.

The complicated case is that there is a mixture of independent and dependent operations in the range of $HB_j [a, j-1]$. Let $EOL = [EO_h, EO_h, ..., EO_h]$ be the list of operations in the range of $HB_j [a+1, j-1]$, which are causally preceding $e_i(O)$. $EOL' = [EO'_h, EO'_h, ..., EO'_h]$, $EO'_h$ is the original form of operation $EO'_h$.

For the first operation in list $EOL$, $EO'_h$ is derived as $EO'_h = \text{LET}(EO_h, HB'_j [a, b+1])$.

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

\[
\begin{align*}
\bullet & \quad TO = \text{LET}(EO_h, HB'_j [a, b+1]); \\
\bullet & \quad EO'_h = \text{IT}(TO, EO'_h).
\end{align*}
\]

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

\[
\begin{align*}
\bullet & \quad TO = \text{LET}(EO'_h, HB'_j [a, b+1]); \\
\bullet & \quad EO'_h = \text{IT}(TO, [EO'_h, EO'_h, ..., EO'_h]).
\end{align*}
\]

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

\[
\begin{align*}
\bullet & \quad TO = \text{LET}(EO_h, HB'_j [a, j-1]); \\
\bullet & \quad O_i = \text{IT}(TO, EOL).
\end{align*}
\]

After each node $s_i$ 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 $s_i$ that, before time $at(r)$, has executed all operations generated at the failed $s_i$ between time $\sigma$ and $\theta$, thus, $\exists 1 \leq j \leq n, j \neq i, r < at(r): \forall O: \sigma < gt_i(O) < \theta \rightarrow e_i(O) \in HB'_j$.

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 $y_i^{\sigma, \delta} = e_i(O)$. For node $s_i (1 \leq j \leq n, j \neq i, y_j^{\sigma, \delta} = e_i(O))$ is the latest operation, where $\delta = at(r)$ is the arrival time of recovery message $r$ from $s_i$ to $s_j$. At time $t_j = at(r)$, $e_i(O)$ is residing in history buffer $HB_i^{\sigma}$ (i.e., $e_i(O) \in HB_i^{\sigma}$) (see also Definition 5). Since the crash recovery algorithm re-arranges operations, which satisfy $s(O) = j \rightarrow (O \Rightarrow y_j^{\sigma, \delta})$, to $s_i: y_i^{\sigma, \delta} \Rightarrow y_j^{\sigma, \delta}$; hence, $y_j^{\sigma, \delta}$ is sent to $s_i$ again. Because $\forall e_i(O) \in HB_i^{\sigma}: s(O) = j \rightarrow (O \Rightarrow y_j^{\sigma, \delta})$ (see Definition 8), we prove that at time $t_j$, $\forall O: gt_i(O) < gt_j(O) < at(r) \rightarrow e_i(O) \in HB_i^{\sigma}$ (see the property of causality preservation). Thus, we obtain

\[
t_a = \max (t_1) = \max (at((O)))
\]

At time $t_a$, we have $\forall O, 1 \leq j \leq n, j \neq i$: \(gt_i(O) < gt_j(O) < at(r) \rightarrow e_i(O) \in HB_i^{\sigma}\). (1)

**Algorithm 4.** The algorithm in $s_i$ to respond to the message $r$. Get $O_o \leftarrow y_i^{\sigma, \delta}$ and $O_h \leftarrow y_j^{\sigma, \delta}$ from the recovery message.  

1. $k \leftarrow 1$
2. $b_i \leftarrow \text{false}$
3. $b_j \leftarrow \text{false}$
4. if $y_i^{\sigma, \delta} = \sigma$ then
5. $b_i \leftarrow \text{true}$
6. end if
7. if $y_j^{\sigma, \delta} = \sigma$ then
8. $b_j \leftarrow \text{true}$
9. end if
10. while $k \leq |HB_i^{\sigma}|$ do
11. $O_o \leftarrow HB_i^{\sigma}[k];$
12. if $b_i = \text{false}$ then
13. if $SV_i = SV_{o_2}$ then
14. $b_i \leftarrow \text{true}$
15. end if
16. else
17. if $S(O) = \sigma$ then
18. send $O' \leftarrow \text{GORT}(O)$ to $s_i$;
19. end if
20. end if
21. if $b_j = \text{false}$ then
22. if $SV_i = SV_{o_2}$ then
23. $b_j \leftarrow \text{true}$
24. end if
25. else
26. if $S(O) = \sigma$ then
27. send $O' \leftarrow \text{GORT}(O)$ to $s_i$;
28. end if
29. end if
30. $k \leftarrow k + 1;$
31. end while

According to assumption 1, let $s_h$ be the node that has executed all operations issued at node $s_i$ between $\sigma$ and $\theta$; thus, we have $\exists 1 \leq i \leq \delta: \forall O: \sigma < gt_i(O) < \theta \rightarrow e_i(O) \in HB_i^{\sigma}$. Therefore, we obtain $\forall O: \sigma < gt_i(O) < \theta \rightarrow e_i(O) \in HB_i^{\sigma}$. Therefore, we obtain $\forall O: \sigma < gt_i(O) < \theta \rightarrow e_i(O) \in HB_i^{\sigma}$. Therefore, we obtain $\forall O: \sigma < gt_i(O) < \theta \rightarrow e_i(O) \in HB_i^{\sigma}$.
Let \( \delta \) be the arrival time of the crash recovery message from \( s_i \) to \( s_i, \delta = at_i(r) \), and \( y^\delta = e_i(LO_i') \) is the latest operation from \( s_i \) in \( HB^\delta \). As described in our algorithm, these operations are delivered back to node \( s_i \) again; we then obtain \( \exists t_p = at_i(LO_i') > \delta : e_i(LO_i') \in HB^\beta \). Because \( \forall e_i(O) \in HB^\delta \), \( s(O) = i \rightarrow (O \Rightarrow LO_i') \), we prove that at time \( t_p \), it is true that \( \forall e_i(O) \in HB^\beta : s(O) = i \rightarrow e_i(O) \in HB^\beta \) (3) (see the causality property).

Based on items (2) and (3) above, we prove that at time \( t_p \), \( \forall O : \sigma < gt(O) < \theta \rightarrow e_i(O) \in HB^\beta \) (4). According to items (1) and (4), we have \( \exists t = \max(t_w, t_p) > \gamma : \forall O : (s(O) = i \land \sigma < gt(O) < \theta) \lor (s(O) = j \neq i \land gt(O_j) < gt(O) < at_i(r)) \rightarrow e_i(O) \in HB^\beta \), where \( y^\beta = e_i(O_j) \). 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.

\[ P(2 \leq i \leq m) \text{ in Figure 2 represents the execution time on a node, it is the nominal measured in CPU cycles between (i-1)th and ith checkpoints.} \]

\[ P_i \text{ indicates the interval between the beginning of the node and its first checkpoint without any transient failure. The total execution time is measured as } P = \sum_{i=1}^{m} P_i. \]

**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 LDS 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. \( f_i(t) \) is the failure probability from the very beginning. Then, we have

\[ C_i = \begin{cases} P_i & \text{with probability } 1 - f_i(P_i) \\ P_i + T_L + C_i & \text{with probability } p \times f_i(P_i) \\ P_i + T_R + C_i & \text{with probability } q \times f_i(P_i) \end{cases} \]  (3)

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

\[ H_i = C_{i-1} + T_L + T_R \]  (4)

\[ H_i = \begin{cases} P_i & \text{with probability } 1 - f_i(P_i) \\ P_i + T_L + C_i & \text{with probability } p \times f_i(P_i) \\ P_i + T_R + C_i & \text{with probability } q \times f_i(P_i) \end{cases} \]  (5)

where \( 2 \leq i \leq m \).

\[ C_i \text{ is derived from Equation 6 as the equation below,} \]

\[ C_i = \frac{C_{i-1} + P_i + (pT_L + qT_R)f_i(P_i)}{1 - f_i(P_i)} \]  (7)

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

\[ C_m = \sum_{i=1}^{m} \prod_{i=j}^{m} \frac{P_i + (pT_L + qT_R)f_i(P_i)}{1 - f_i(P_i)} \]  (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 \( p^L \) and \( p^U \) be the probability of the ith LDS approach becoming successful and unsuccessful, respectively, where \( p^L + p^U = 1 \). \( C^L(P, k) \) is given as below,

\[ C^L(P, k) = (P + T_L) p^L + 2(P + T_L) p^U p^L + \ldots + k(P + T_L) \]

\[ \sum_{i=1}^{k} \left[ (P + T_L) p^U \right] + \left[ k(P + T_L) \right] \prod_{i=1}^{k} p^U \]  (9)

\[ = \sum_{i=1}^{k} \left[ j + (P + T_L) \right] \prod_{i=1}^{k} p^U + k(P + T_L) \]
and 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,

\[
p_i^u < p_{i+1}^u < \cdots < p_i^s \quad \text{and} \quad p_i^s < p_{i+1}^s < \cdots < p_k^s
\]

(10)

We assume that \(p_i^s = w_i < 1\), and for the simplicity, it is assumed that \(w_1 = w_2 = \ldots = w_k = w\), and \(p_k^u = p\). Equation 11 is derived from Equation 10 as follows.

\[
CL(P, k) = \sum_{i=1}^{k-1} (1-p^{w-1}) + k(P+T_L) \prod_{j=1}^{i-1} (1-p^{w-1}) + k(P+T_L) \prod_{j=i}^{k-1} (1-p^{w-1})
\]

(11)

The time overhead of LDS recovery is determined by \(P\) and the arrival rate of operations \(\lambda\). 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,

\[
C^R(R) = \frac{P+T_s}{1-f_r(P)}
\]

(12)

LDS crash recovery is an efficient method to recover the temporary failures in nodes 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 \(P\), \(C^R(P, k)\) can be determined by \(k\), which must satisfy \(C^R(P, k) < C^S(P)\).

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

Table 2. \(P = 100, T_L= 20, T_R = 40, w = 0.8, f_r(P) = 0.1\)

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

To evaluate the impact of the probability of the first successful LDS recovery on \(C^R(P, k)\), we fix \(T_L, T_R, w,\) and \(f_r(P)\), and increased \(k\) from 10 to 30 with an increment of 10. Table 2 shows the execution time of the node in the presence 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 3 illustrates the relation between \(w\) and \(C^R(P, k)\). \(T_L, T_R, p,\) and \(f_r(P)\) are fixed, and \(k\) is set to 10, 20, and 30, respectively. Like the effect of \(p\) on \(C^R\), 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, \(C^R\) 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. \(P = 100, T_L= 20, T_R = 40, w = 0.8, f_r(P) = 0.1\)

<table>
<thead>
<tr>
<th>(w)</th>
<th>0.65</th>
<th>0.70</th>
<th>0.75</th>
<th>0.80</th>
<th>0.85</th>
<th>0.90</th>
</tr>
</thead>
<tbody>
<tr>
<td>(k=10)</td>
<td>208.5</td>
<td>192.9</td>
<td>179.2</td>
<td>166.7</td>
<td>154.8</td>
<td>143.3</td>
</tr>
<tr>
<td>(k=20)</td>
<td>217.4</td>
<td>197.9</td>
<td>181.8</td>
<td>168.0</td>
<td>155.4</td>
<td>143.4</td>
</tr>
<tr>
<td>(k=30)</td>
<td>234.3</td>
<td>208.5</td>
<td>188.2</td>
<td>171.6</td>
<td>157.3</td>
<td>144.3</td>
</tr>
</tbody>
</table>

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 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

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-real-time</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Real-time</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fault tolerance</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Consistency maintenance</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fail recovery</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>

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 differencing 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 Mojcn 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

pp. 675-678.


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.

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.

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.