PAPER

A Concurrent Partial Snapshot Algorithm for Large-Scale and Dynamic Distributed Systems

Yonghwan KIM[†a)], ***Nonmember***, **Tadashi ARARAGI**[††b)], ***Member***, **Junya NAKAMURA**[†c)], ***Nonmember***, ***and*** **Toshimitsu MASUZAWA**[†d)], ***Member***

SUMMARY Checkpoint-rollback recovery, which is a universal method for restoring distributed systems after faults, requires a sophisticated snapshot algorithm especially if the systems are large-scale, since repeatedly taking global snapshots of the whole system requires unacceptable communication cost. As a sophisticated snapshot algorithm, a partial snapshot algorithm has been introduced that takes a snapshot of a subsystem consisting only of the nodes that are communication-related to the initiator instead of a global snapshot of the whole system. In this paper, we modify the previous partial snapshot algorithm to create a new one that can take a partial snapshot more efficiently, especially when multiple nodes concurrently initiate the algorithm. Experiments show that the proposed algorithm greatly reduces the amount of communication needed for taking partial snapshots.

key words: *fault-tolerance, large-scale distributed system, concurrent snapshot, checkpoint, rollback*

1. Introduction

A distributed system consists of computational entities (i.e., computers, usually called nodes) that are autonomous and connected each other by asynchronous communication links [5]. Nodes communicate with each other by exchanging messages. Because distributed systems are prone to faults, fault tolerance of distributed systems is a key goal.

Checkpoint-rollback recovery [7], [13], [16] is a universal method for restoring a distributed system from faults. Each node periodically records its local state into non-volatile storage from which it recovers its state when faults occur. The stored state of a node is called a (local) *checkpoint*, and restoring the node state to the checkpoint is called a *rollback*.

From a system-wide viewpoint, each node has to recover its state so that the states of all nodes form a *consistent* global state [17] (or a global state without an orphan message, which is a received message but does not be sent by the sender). To avoid the domino effect of rollbacks [23] (or an unbounded chain of rollbacks for attaining a consistent global state), nodes have to store their states cooperatively so that the stored checkpoints form a consistent global state of the whole system. The consistent global state formed by the recorded checkpoints of the nodes is called a *snapshot*, and the algorithm with which nodes record their checkpoints cooperatively is called a *snapshot algorithm*.

Many sophisticated snapshot algorithms have been proposed. As the size of a distributed system increases, the efficiency of snapshot algorithms becomes more important. In large-scale distributed systems, repeatedly taking global snapshots of the whole system incurs unacceptable communication cost. Another important requirement of snapshot algorithms is adaptability to dynamic distributed systems like web services [10], for example, RosettaNet [12], where nodes can freely join and leave the system at any time.

Related works: Chandy and Lamport [1] proposed a distributed algorithm for taking a (global) snapshot of a whole system. This snapshot algorithm assumes that all the channels guarantee the FIFO property. Lai and Yang [20] and Mattern [19] proposed snapshot algorithms for distributed systems with non-FIFO channels. All the above algorithms allow easy implementation and high efficiency, but still require $O(n^2)$ messages since every pair of neighboring nodes exchanges messages, where n denotes the number of nodes in the system. Thus, these algorithms are not applicable, in practice, to large-scale distributed systems. While further research reduces complexity by simplifying the taking of snapshots (e.g. $O(n \log n)$) [3], [6], [20]–[22], the scalability of snapshot algorithms remains critical. Moreover, it is impossible to apply these algorithms to dynamic distributed systems where nodes can join and leave the system at any time, which is common in a system like web-services.

Communication-induced checkpointing is an alternative approach to scalable snapshot algorithms [24]–[27]. Not all nodes are requested to record their checkpoints in the algorithms, but some are, depending on the communication pattern. For distributed applications based mainly on the local coordination of nodes, communication-induced checkpoint algorithms can reduce the communication and time required for recording the checkpoints of nodes. However, the recorded checkpoints provide no guarantee that a consistent global state can be formed from the latest checkpoints of the nodes. This forces each node to keep multiple checkpoints in its non-volatile storage and requires a sophisticated method to find a set of checkpoints of nodes that forms a consistent global state.

Manuscript received February 27, 2013.
Manuscript revised August 31, 2013.

[†]The authors are with the Graduate School of Information Science and Technology, Osaka University, Suita-shi, 565–0871 Japan.
[††]The author is with NTT Communication Science Laboratories, NTT Corporation, Kyoto-fu, 619–0237 Japan.
a) E-mail: y-kim@ist.osaka-u.ac.jp
b) E-mail: araragi.tadashi@lab.ntt.co.jp
c) E-mail: junya-n@ist.osaka-u.ac.jp
d) E-mail: masuzawa@ist.osaka-u.ac.jp

DOI: 10.1587/transinf.E97.D.65

Copyright © 2014 The Institute of Electronics, Information and Communication Engineers

To achieve scalability of snapshot algorithms, Moriya and Araragi [2], [3] introduced a *partial snapshot*[*] of a subsystem consisting only of communication-related nodes and showed that the whole system can recover from faults by restoring each node of the subsystem to the state in only the latest partial snapshot. For taking a partial snapshot, Moriya and Araragi [2], [3] also proposed a distributed algorithm called the *Sub-SnapShot (SSS)* algorithm. In practical distributed applications, the number of nodes in a communication-related subsystem is expected to be much smaller than the total number of nodes. Therefore, SSS algorithm can take a partial snapshot efficiently and thus makes the checkpoint-rollback recovery applicable to large-scale distributed systems. Another important advantage of SSS algorithm is that it is applicable to dynamic distributed systems (where nodes can join and leave the system), because each node needs no a priori knowledge of its adjacent nodes and augments the knowledge during the execution of SSS algorithm. Koo and Toueg [13] introduced a communication-induced checkpoint algorithm applicable to dynamic distributed systems, but it has to suspend the executions of applications while taking checkpoints to guarantee consistency. On the other hand, SSS algorithm frees itself from this suspension by an elaborate operation on communication-relation. Lastly, SSS algorithm satisfies the strong consistency requirement just as Chandy and Lamport [1] does. That is, if a sender node records sending of a message in its stored local state, then the receiver node must record the message as a received or an in-transit one.

Although SSS algorithm has the above advantages, its drawback is an inability to guarantee the strong consistency of the partial snapshot it takes when multiple nodes concurrently initiate SSS algorithm and the subsystems communication-related to the initiators overlap. Each node may need to confirm its local state at any time independently from other nodes and this causes that two or more nodes initiate the snapshot algorithms (especially in the large-scale distributed system) at the same time. Thus, resolving this problem (two or more snapshot algorithms overlap) is necessary to apply a partial snapshot algorithm to large-scale distributed system. We call this problematic situation a *collision* of SSSs. The collision of concurrent SSS algorithms can be obviously resolved when concurrent initiations are handled sequentially one by one, but algorithm efficiency is severely degenerated. Spezialetti [14] and its improved variant by Prakash [15] solved the problem of concurrent initiations. But their methods still target the creation of a global snapshot, and while it may be partitioned into pieces for efficiency, they are not applicable to dynamic situations.

Contribution of this paper: In this paper, we resolve the collision problem in SSS algorithm by presenting a new partial snapshot algorithm called *Concurrent Sub-Snapshot (CSS)* algorithm that can efficiently take a partial snapshot even when multiple nodes concurrently initiate the algorithm. CSS algorithm successfully realizes an efficient solution for the collision problem by consistently merging two concurrent SSS algorithm executions when a collision occurs on a node. The initiators of the concurrent executions communicate with each other to cooperatively combine the executions, but the overhead for the combining process is small. The experimental results prove that CSS algorithm requires very small cost for resolving the collision problem, compared with naive solutions.

The rest of this paper is organized as follows: Sect. 2 introduces our system model and defines the problem. Section 3 presents our algorithm and experimental evaluations are presented in Sect. 4. A conclusion is given in Sect. 5.

2. Preliminaries

2.1 System Model

In this paper, we consider distributed systems consisting of nodes (or processes). Nodes share no common memory or storage and communicate with each other asynchronously by exchanging messages through the communication channels. Each node has its own local state which is characterized it's initial state and operation history [9]. An enormous number of nodes can exist in the (large-scale) system however the number of nodes is assumed to be finite. This implies that a distributed algorithm which requires the communication spread over all the nodes in the system is not practical.

We assume the distributed systems are dynamic [28], where nodes can frequently join and leave the system. This means that the network's topology is changeable and each node never recognizes the entire system's configurations.

Each node has a comparable unique identifier (ID) drawn from a totally ordered set. The system is a fully connected network where all the nodes can directly communicate with all other nodes. All links used for the message transfer are reliable and FIFO, so all the messages are eventually received and the messages sent using the same link (in the same direction) are received in the order they were sent. We assume an asynchronous distributed system, where messages experience finite but unpredictable delay.

We also assume that distributed systems considered in this paper are applicable to checkpointing-rollback paradigm. This means that the distributed systems can restart executions from the restored snapshots, and thus we exclude the distributed systems with real-time interactions or human-interactions.

2.2 SSS Algorithm

Our work, CSS algorithm, is based on SSS algorithm that creates a partial snapshot of a subsystem [2], [3]. We briefly explain SSS algorithm before introducing CSS algorithm (Sect. 3). SSS algorithm has the following properties. (a) Snapshots are only created among communication-related

[*]In [14], part of a global snapshot is also called a *partial snapshot*, but the notion is completely different from that in SSS algorithm, which is not a part of a single static global snapshot.

nodes so that a rollback can only be done for the nodes related to a crashed node. Therefore, even if there are plenty of nodes in a distributed system, SSS algorithm can efficiently make a partial snapshot, and effectively achieve a rollback. (b) SSS algorithm is applicable to dynamic distributed systems (where nodes can join and leave freely), because each node doesn't have to know the set of nodes in the system beforehand. Since a set of nodes contained in a partial snapshot is dynamically determined during the execution of SSS algorithm, it can cope with dynamic joining and leaving of nodes.

To achieve the above properties, SSS algorithm traces the communication-relation of the application algorithm: when a node records its checkpoint, it only requests nodes with which it has communicated to record their checkpoints. To realize this strategy, each node $node_i$ maintains a set DS_i, called a dependency set, of nodes with which it has communicated (sent or received messages). The set determines neighbors of the subsystem for which a partial snapshot should be taken. When a node $node_i$ receives an application message from another node, say $node_j$, or sends one to $node_j$, then the ID of $node_j$ is added to DS_i. DS_i always includes its own ID, $node_i$.

When a node $node_i$ initiates SSS algorithm, it records its local state and sends a marker with its ID to the nodes in DS_i at that moment. This node is called an *initiator*. When a node $node_j$, which is not an initiator, receives a marker for the first time, it records its local state and forwards the marker to each $node_x$ in DS_i. By this way, the marker is forwarded to all nodes in the subsytem for which a partial snapshot should be taken. On receipt the marker, $node_j$ sends a pair of its DS contents and ID $(DS_j, node_j)$ to the initiator, which maintains the union of the received dependency sets (including its own) and the set of received IDs (including its own). When the union of the dependency sets and the set of IDs become equal, the initiator decides that the nodes in the set constitute the subsystem where the partial snapshot should be taken. We call this a *partial snapshot group* (dependency relation group, DRG). Based on this set, the initiator informs each node $node_j$ of the set DRG_j of nodes from which $node_j$ should receive markers. This set can be defined as follows: $DRG_j = \{ node_x \mid node_j \in DS_x \}$

Recording application messages and finishing the snapshot algorithm are done in the same way as traditional algorithms. Like as Chandy's algorithm, SSS algorithm allows the application algorithm to continue running during execution of the snapshot algorithm. To guarantee consistency, a node has to send a marker to another node that isn't contained in its DS before sending an application message. When SSS algorithm is terminated, each node in DRG clears its DS. With these procedures, SSS algorithm efficiently allows a small portion of a consistent snapshot efficiently. Thus, SSS algorithm can make the checkpoint correctly, even in a dynamic distributed system where the network topology is never statically fixed. However, as stated in Sect. 1, SSS algorithm does not guarantee consistency if there is a collision in a node. Our proposed algorithm solves this problem without sacrificing efficiency.

2.3 Problem Definition

Moriya and Araragi [2], [3] assume that two or more SSS algorithms are NOT simultaneously executed on the same node to guarantee the consistency of the partial snapshot. However, in some distributed applications, each node may need to initiate the snapshot algorithm independently from other nodes in order to confirm its current local state and commit the result of the requests. If each node can initiate the snapshot algorithm (become an initiator) at any time, SSS algorithm hardly guarantees its consistency because two or more snapshot algorithms may be collided each other. A naive solution to such a problem is that an initiator of the snapshot algorithm informs every node in the system of its initiation (i.e. broadcasting). However, the solution is not practical, especially in a large-scale distributed system. We define a *collision of SSS algorithms on a node* to be a situation where two or more SSS algorithms are executed at the same moment on the node.

Now we explain the problem of a collision in SSS algorithm. Here we assume that two or more SSS algorithms are operated independently and correctly. And that, each node maintains only the latest checkpoint. Therefore, when each node concurrently engages in two or more executions of SSS algorithm, it will keep only the one of the checkpoints created by the executions. Figure 1 shows two different executions, where only the behaviors of two nodes $node_a$ and $node_b$ among many nodes in the system are illustrated.

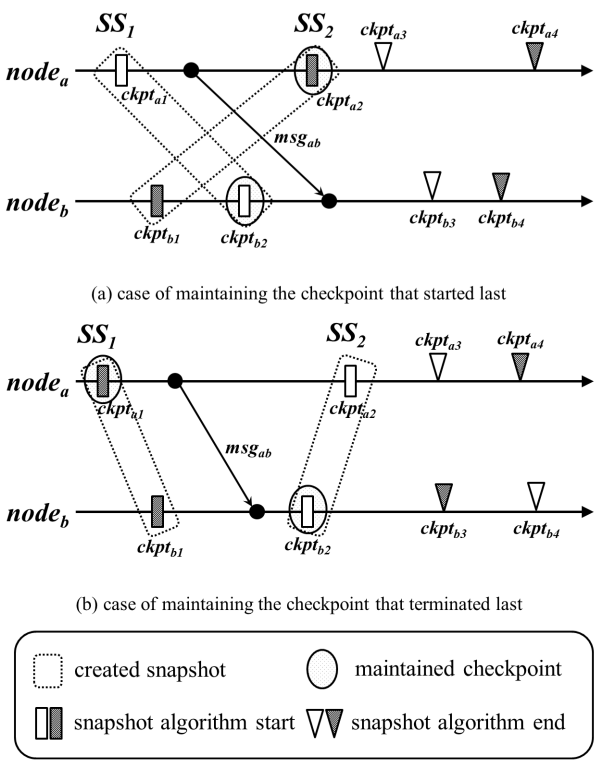

Fig. 1 Concurrent executions of SSS algorithm.

In the both executions, collisions occur at the two nodes between two SSS algorithm SS_1 and SS_2. First, we consider the case that each node maintains the checkpoint recorded by the execution that *started* the most recently (Fig. 1 (a)). In this case, $node_a$ maintains $ckpt_{a2}$ of SS_2 and $node_b$ does $ckpt_{b2}$ of SS_1. Thus, if $node_a$ and $node_b$ rollback for recovery, $node_a$ returns to checkpoint $ckpt_{a2}$ and $node_b$ does to checkpoint $ckpt_{b2}$. Sending of message msg_{ab} is recorded in $ckpt_{a2}$, however, msg_{ab} is NOT recorded (even as an in-transit message) in $ckpt_{b2}$ since SS_1 takes a consistent partial snapshot and sending of msg_{ab} is not recorded in $ckpt_{a1}$. This violates the strong consistency of the snapshot algorithm as mentioned in Sect. 1. Second, we consider the case that each node maintains the checkpoint recorded by the execution that *terminated* the most recently (Fig. 1 (b)). In this case, $node_a$ maintains $ckpt_{a1}$ and $node_b$ does $ckpt_{b2}$. Because SS_1 is terminated at $ckpt_{a4}$ in $node_a$ and SS_2 is terminated at $ckpt_{b4}$ in $node_b$. By the observation similar to the first case, $ckpt_{b2}$ records the receipt of msg_{ab} but $ckpt_{a1}$ does not record the sending of msg_{ab}. This causes msg_{ab} to be an orphan, which violates the consistency of the snapshot.

To guarantee the consistency, one possible approach is to modify the rollback algorithm so that the nodes do not necessarily return to their last checkpoints but return to some adequate checkpoints to avoid such inconsistency. However, we need a sophisticated method for finding adequate checkpoints. In addition, each node has to keep multiple checkpoints, up to the number of nodes, while Chandy's algorithm and SSS algorithm require each node to store only the latest checkpoint it takes. Thus, we lose the efficiency achieved by Chandy's algorithm and SSS algorithm. In this paper, we take another approach. We modify SSS algorithm so that concurrent SSS algorithm executions in collision are combined to a single SSS algorithm execution. With this merging, recording the latest checkpoint at each process is sufficient and the original efficient rollback procedure can be utilized. As we mentioned previously, we call our partial snapshot algorithm *Concurrent Sub-Snapshot (CSS)* algorithm.

3. Concurrent Partial Snapshot Algorithm

In this section, we introduce CSS algorithm that can combine the concurrently executed SSS algorithms when a collision occurs.

3.1 Overview of CSS Algorithm

CSS algorithm solves the collision problem by combining two or more colliding SSS algorithms. In the CSS algorithm, a representative node, called a *main-initiator*, is selected among the initiators of the colliding SSS algorithms. Then the main-initiator behaves as an initiator of the combined SSS algorithms.

Figure 2 (a) shows two partial snapshot groups concurrently executing CSS algorithm. Each group includes a unique initiator and consists of nodes communication-related to the initiator and each snapshot group is NOT fixed currently. If there is a communication-relation between two nodes that belong to different partial snapshot groups, the collision will occur between these two groups. Note that either this communication-relation can be created after initiating the snapshot algorithm, or this communication-relation can already exist in the same snapshot group that is initiated by two or more different initiators. The node that is related to another group's node recognizes the collision when receiving a new marker with different initiator ID. The node that detects the collision informs its group's initiator (this informing message will be forwarded to main-initiator of the group). If the initiator has not determined its partial snapshot group yet, the two executions of the SSS algorithm are combined. The initiator directly communicates with the other initiator to decide which initiator should be the main initiator of the combined SSS algorithm: the initiator with a prior ID becomes the main initiator. The other initiator is called a *sub-initiator*. The sub-initiator generates a logical link to the main-initiator that is called a *main link*. This process is depicted on Fig. 2 (b). After the selection of the main-initiator, the sub-initiator passes the information (DSs and node IDs) it has collected and forwards incoming information to the main-initiator.

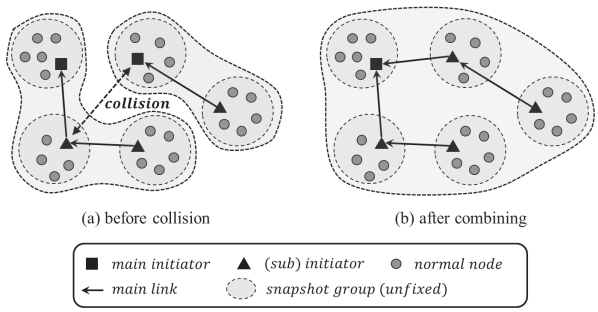

Fig. 2 Combining of two executions of SSS algorithm.

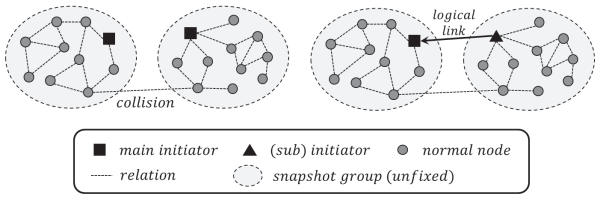

Fig. 3 Combining of two executions of SSS algorithm.

If another collision occurs between the combined SSS algorithm and another SSS algorithm, the combining is performed exactly in the same way, as depicted in Fig. 3. Note that the local state to be stored as a checkpoint of a collided node is the one at the time when the node received a marker for the first time. This way of storing achieves consistency and avoids deadlock. In the selection of the main-initiator, an additional procedure is performed (not mentioned here) to avoid deadlock. The details are explained in Sect. 3.3.

Our previous work, SSS algorithm, guarantees the efficiency of taking snapshot even if the system is large-scale,

because SSS algorithm should be executed on the small subsystem consisting of communication-related nodes. This implies that the complexity of the snapshot algorithm is independent from the scale (the number of nodes) of the whole system. CSS algorithm, based on SSS algorithm, also preserves this property obviously. However, even the scale of the subsystem is same, higher efficiency might be expected by CSS algorithm because it can take the snapshot of the subsystem concurrently with two or more initiators.

3.2 CSS Algorithm

In this section, we give the pseudo codes of CSS algorithm.

Variables of each node
init : Stores an initiator's ID. The value is initially $\perp$. When a node receives a marker and the value of *init* is $\perp$, then it is set to the initiator ID of the marker. When the execution of CSS algorithm on the node terminates, init is initialized to $\perp$ (for the next execution of CSS algorithm).
MainLink : Stores the main-initiator's ID. The value is initially $\perp$. When the node initiates CSS algorithm, then the value is set to its own ID. When the node becomes a sub-initiator after it combines with another execution of CSS algorithm, the MainLink is set to the ID of the main-initiator.
WaitFlag : A Flag variable to consistently select a main-initiator when the collision occurs.
RcvMkList : This set variable stores the IDs of the nodes from which the node received markers.
DS : This set variable stores the IDs of the nodes in its Dependency Set.
DSSender : This set variable is used only by an initiator to store the IDs of the nodes from which DS information was received.
MustRcv : This set variable stores the IDs of the nodes from which the node has to receive markers (before termination).
AllDS : This set variable is used only by an initiator to store the union of the DSs it received.
MsgQ : This message queue stores the messages received during the process to resolve the collision. The stored messages are processed after the collision is resolved.
FinFlag : A flag variable to determine termination. Value will be TRUE when Fin message is received.
msg : The set variable to store currently received messages. This variable includes not only message's type but also attached parameters and sender's ID.

We use one constant variable, **self**, which stores each node's own ID.

Messages: Messages for CSS algorithm have the form of (< $Message\_type$, Parameters>, Sender). The message types are listed below. The Sender is the ID of the sender. We introduce the following three types of messages for taking a snapshot and an additional five types of system messages for handling a collision.
<**Marker, Initiator**> Marker message. Parameter *Initiator* denotes the initiator's ID.
<**DSinfo, localDS, DSfrom**> A message to send its own DS (all the nodes communication-related to this node) to the initiator designated in the marker.
<**Fin, DRGinfo**> Termination message. Parameter $DRGinfo$ is the node list from which each node has to receive markers (before termination).
<**NewInit, CollidedNode, 2ndInit**> A informing message of detecting a different snapshot algorithm. The message is delivered to the main-initiator from a collided node (or the node detecting a collision). $CollidedNode$ is the collided node's ID, and $2ndInit$ is the ID of the initiator that caused the collision (the ID is contained in the marker that caused the collision). If the initiator (the destination of *NewInit* message) has already become a sub-initiator, the message is relayed to the main-initiator using *MainLink*.
<**Accept, Initiator, 2ndInit**> A message which implies combining is possible. When a main-initiator receives a *NewInit* message and decides that these two SSS algorithms should be combined, it sends *Accept* message to the collided node.
<**Combine, CollidedNode, 1stInit**> A informing message of combining two different snapshot algorithms. When a collided node receives an *Accept* message, it sends *Combine* message to the node that sent *Marker* message that caused the collision. In the same way as *NewInit* message, if the destination has already become a sub-initiator, it relays the message to the main-initiator using *MainLink*.
<**CompInit**> A message for compare the priority between two initiators' ID. When a main-initiator receives *Combine* message, it sends *CompInit* message directly to the other main-initiator to compare their IDs.
<**InitInfo, DSInfo, SenderInfo**> A message from a sub-initiator for notifying its information (including DS maintained) of a main-initiator. When two algorithms are combined, the main-initiator with low priority (now becomes a sub-initiator) sends $InitInfo$ message to forward the DS information and the DS senders' list it collected to the main-initiator with high priority.

Algorithm 1 and 2 represent the pseudo codes of CSS algorithm. Figure 4 shows the flow of the messages from *Marker* to *CompInit* when a collision occurs between two different executions of CSS algorithm.

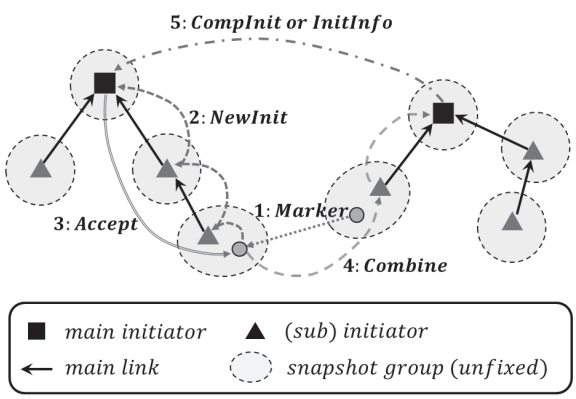

Fig. 4 Message flow for handling a collision in CSS algorithm.

Algorithm 1 Pseudo Code of CSS Algorithm

```
 1: procedure INITIALIZE()                          ▷ Initialize Snapshot Algorithm
 2:     MainLink ← self
 3:     send(self, <Marker, self>)                  ▷ send Marker to itself
 4: end procedure

 5: procedure ONRECEIVE(<Marker, ?init>, ?sender)
                                                    ▷ when Marker is received
 6:     if init = ⊥ then                            ▷ receipt of the first Marker
 7:         Store Local State
 8:         init ← ?init
 9:         RcvMkList ← RcvMkList ∪ { ?sender }
10:         send(?init, < DSinfo, DS>)  ▷ send current DS to the initiator
11:         send(∀node_i ∈ DS, <Marker, ?init>)     ▷ broadcast Marker
12:     else if init = ?init then                   ▷ second or later Marker
13:         RcvMkList ← RcvMkList ∪ { ?sender }
14:         if FinFlag then
15:             TerminationCheck()
16:         end if
17:     else                                        ▷ collision has occurred
18:         MsgQ.enqueue(msg)      ▷ store Marker to the message queue
19:         send(init, <NewInit, self, ?init>)
                                                    ▷ notify the initiator of collision
20:     end if
21: end procedure

22: procedure ONRECEIVE(<DSinfo, ?localDS, ?DSfrom>, ?sender )
                                                    ▷ when DSinfo is received
23:     if MainLink = self then                     ▷ case of the main-initiator
24:         AllDS ← (AllDS ∪ ?localDS)              ▷ to update group node list
25:         DSSender ← DSSender ∪ ?DSfrom ▷ record sender node's ID
26:         if !WaitFlag then
27:             TerminationCheck()  ▷ cannot terminate during combining
28:         end if
29:     else                          ▷ message forwarding to the main-initiator
30:         send(MainLink, <DSinfo, ?localDS, ?DSfrom>)
31:     end if
32: end procedure

33: procedure ONRECEIVE(<Fin, ?DRGinfo>, ?sender)
                                                    ▷ when Fin is received
34:     MustRcv ← DRGinfo
35:     FinFlag ← TRUE                              ▷ FinFlag is ON
36:     TerminationCheck()
37: end procedure

38: procedure ONRECEIVE(<NewInit, ?node_i, ?init>, ?sender)
                                                    ▷ when NewInit is received
39:     if MainLink = self then                     ▷ case of main-initiator
40:         if WaitFlag then                        ▷ waiting condition
41:             MsgQ.enqueue(msg)
42:         else if DSSender ⊂ AllDS then   ▷ group is not decided yet
43:             send(?node_i, <Accept, self, ?init>)  ▷ permit for combine
44:             DS ← ( DS ∪ ?init )
45:             WaitFlag ← TRUE
46:         end if
47:     else                          ▷ message forwarding to the main-initiator
48:         send(MainLink, <NewInit, ?node_i, ?init>)
49:     end if
50: end procedure                                   ▷ continue to Algorithm 2
```

Algorithm 2 Pseudo Code of CSS Algorithm (Cont.)

```
51: procedure ONRECEIVE(<Accept, ?init_i, ?init_j >, ?sender)
52:                                                 ▷ when Accept is received
53:     receive( MsgQ.Dequeue( msg(<Marker, ?init_q >, ?node_q ) )
54:     send(?init_j, <Combine, self, ?init_i > );
55:             ▷ notifying collision and combining to opponent initiator
56:     send(?node_q, <Marker, ?init_q > )           ▷ reply marker
57: end procedure

58: procedure ONRECEIVE(<Combine, ?node_k, ?init>, ?sender)
59:                                                 ▷ when Combine is received
60:     if MainLink = self then                     ▷ case of the main-initiator
61:         if self < ?init then        ▷ self is prior to the opponent initiator
62:                 ▷ WaitFlag is ignored to resolve the deadlock problem
63:             send(?init, <CompInit> )
64:                             ▷ the opponent will become sub-initiator
65:         else                          ▷ opponent initiator is prior to self
66:             if WaitFlag then                    ▷ WaitFlag is ON
67:                 MsgQ.Enqueue(msg)
68:                             ▷ wait for the previous combine process
69:             else                          ▷ case of being sub-initiator
70:                 send(?init, <InitInfo, AllDS, DSSender>)
71:                 MainLink ← ?init
72:             end if
73:         end if
74:     else                                  ▷ forwarding message to MainLink
75:         send( MainLink, <Combine, ?node_k, ?init> )
76:     end if
77: end procedure

78: procedure ONRECEIVE(<CompInit>, ?sender )
                                                    ▷ when CompInit is received
79:     send( ?sender, <InitInfo, AllDS, DSSender> )
80:     MainLink ← ?sender
81:     WaitFlag ← FALSE
82:     Process the messages in MsgQ
83: end procedure

84: procedure ONRECEIVE(<InitInfo, ?AllDS, ?DSSender>, ?sender)
                                                    ▷ when InitInfo is received
85:     AllDS ← (AllDS ∪ ?AllDS)
86:     DSSender ← (DSSender ∪ ?DSSender)
87:       ▷ apply opponent's DS and Senders to maintain combined group
88:     WaitFlag ← FALSE
89:     TerminationCheck()
90: end procedure

91: procedure TERMINATIONCHECK()
92:     if MainLink=self then                       ▷ case of the main-initiator
93:         if AllDS ⊆ DSSender then
94:             send(∀node_i ∈ AllDS, <Fin, {node_j | node_j sent the marker
    to node_i} >)
95:         end if
96:     else                                  ▷ all other nodes except main-initiator
97:         if MustRcv ⊆ RcvMkList then
98:             terminate algorithm
99:         end if
100:    end if
101: end procedure
```

In pseudo codes, lines 1 to 3 represent the initiation of the snapshot algorithm. Each node can be an initiator by sending *Marker* to itself. Note that procedure *Initialize*() can be executed at any time by any nodes.

Lines 4 to 20 show the procedure executed when the *Marker* is received. If the node which received the *Marker* has no initiator (line 6), it stores its local state to stable storage and broadcasts *Marker* messages to its communication-related nodes. After broadcasting, the node should receive the replying *Markers* of broadcasted *Markers* (line 12). These replying *Markers* will be recorded (line 13) to de-

termine termination of the snapshot algorithm.

Collision of two snapshot algorithms is found when the node received *Marker* with an attached different initiator ID (line 17). The node which detects the collision notifies its original initiator (ID stored in variable *init*) of collision and opponent initiator's ID by *NewInit* message (line 19). This *NewInit* message will be forwarded through *MainLink* (line 48) if the initiator received the *NewInit* message is not a main-initiator. The main-initiator which received *NewInit* message checks whether DRG is fixed or not (line 42). Remember that DS_i includes $node_i$ itself (mentioned in Sect. 2.2), thus DSSender is proper subset of the AllDS unless all nodes in AllDS send their DS to the initiator. If DRG is not fixed, the initiator sends *Accept* message to combine the two collided snapshot algorithms. Note that the opponent initiator never terminates its snapshot algorithm because the node which detects the collision didn't send any replies to received markers. Moreover, the initiator which sends *Accept* message to the opponent initiator cannot terminate the snapshot algorithm without receipt of a replying message of *Accept* message(*CompInit* or $InitInfo$ message) because it adds the opponent initiator's ID to its DS (line 44) and *WaitFlag* is true (line 45).

The node that received *Accept* message sends *Combine* message to the *Marker*'s sender (lines 51 to 57). Same as *NewInit* message, *Combine* message will be forwarded through *MainLink* (line 75) if the initiator that received the *Combine* message is not a main-initiator. Unless *WaitFlag* is true, the main-initiator compares its own ID's priority and the opponent's priority. If its ID is prior to the opponent's ID, *CompInit* message should be sent. If not, $InitInfo$ should be sent and its *MainLink* should be updated to the opponent's ID. However, even though *WaitFlag* is true, if its own ID is prior to the opponent ID (line 61), it sends *CompInit* message to the opponent to aviod deadlock (details will be mentioned in Sect. 3.4).

Procedure $TerminationCheck()$ (line 91 to 101) should be executed for checking termination of the snapshot algorithm. After terminating algorithm (line 98), all variables will be initiated to the initial values and DS information before initiating snapshot algorithm will be cleared.

3.3 Technical Discussion of CSS Algorithm

In this section, we give proof sketches for the correctness of CSS algorithm. First, we show that it will eventually terminate without deadlock. Next, we prove that the partial snapshot obtained by it is consistent.

Deadlock Problem and Solution

In CSS algorithm, node IDs are compared among main-initiators to decide a new main-initiator for the combined partial snapshot. However, if two combining procedures occur on a main-initiator in parallel, and the initiator becomes a sub-initiator for one combining before the *Combine* message reaches the main-initiator of the other, then it causes an inconsistent situation. To prevent this problem, the main-initiator that permits combining must not become a sub-initiator by another CSS algorithm until the *CompInit* message is received. However, the introduction of this waiting condition can cause a deadlock. For instance, in Fig. 5, three main-initiators are waiting for replies from each other. To solve this deadlock problem, CSS algorithm uses an exception handling of a waiting condition. Even if a main-initiator is in a waiting condition, it replies with the *CompInit* message to the opponent main-initiator if the opponent's ID priority is lower than itself.

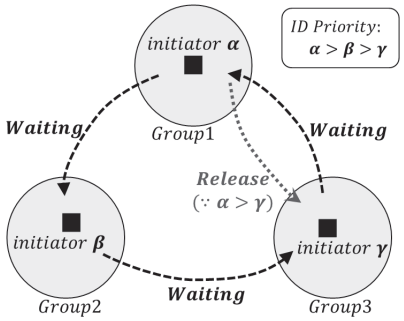

Fig. 5 Example of deadlock.

In Fig. 5, three main-initiators enter the waiting state and wait one another. In this case, at least one main-initiator has a higher priority than the its waiting main-initiators. For example, let the main-initiators of groups 1, 2, and 3 have IDs α, β, and γ respectively, where the priority level is $\alpha > \beta > \gamma$. In this case, initiator α receives the *Combine* message from initiator γ and initiator α learns that initiator γ should become a sub-initiator after combining. After initiator α sends CompInit to initiator γ, initiator γ leaves its waiting condition and the deadlock is released. In this manner, if such a waiting state occurs, at least one node eventually receives *Combine* message from a main-initiator with a lower priority. Therefore the deadlock is avoided by this exception handling operation.

Snapshot Consistency

Theorem 1. *The combined partial snapshot obtained by CSS algorithm is consistent.*

Proof sketch. We show that any partial snapshot obtained by CSS algorithm can also be obtained by SSS algorithm executed for distributed systems running the same application. Because the consistency of the partial snapshot by SSS algorithm has already been proven in [2], [3], we deduce that the partial snapshot by CSS algorithm is consistent. For any given execution of CSS algorithm, we can construct an execution of SSS algorithm that obtains the same partial snapshot. To help understanding, we show a construction example that deals with the messages employed in partial snapshots: Application messages, markers, dependent set information (DS), termination messages and system messages for handling the collision.

Consider the execution of CSS algorithm in Fig. 6. Each $node_i$ starts a snapshot algorithm at $ckpt_{i1}$ and ter-

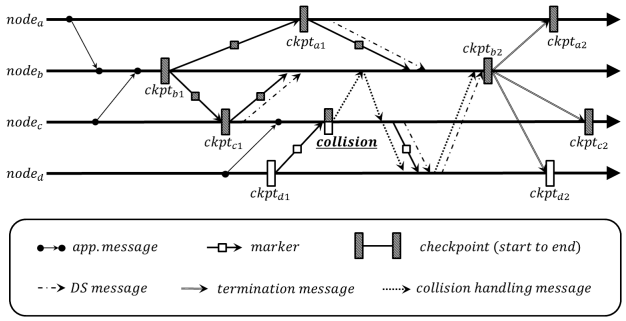

Fig. 6 Execution example of CSS algorithm.

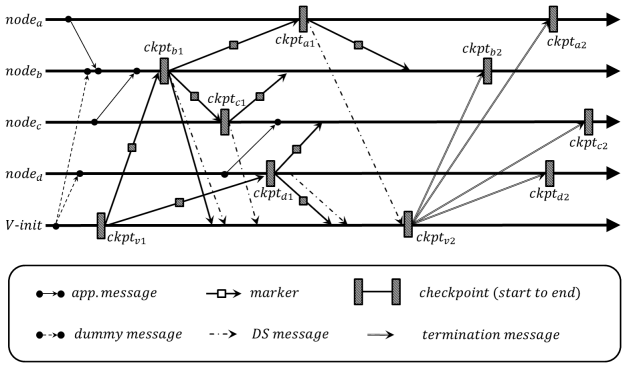

Fig. 7 SSS algorithm simulation.

minates taking a checkpoint at $ckpt_{i2}$. In this execution, two partial snapshot algorithms are individually initiated at $ckpt_{b1}$ of $node_b$ and $ckpt_{d1}$ of $node_d$. They collide on $node_c$ and are combined. In the combining, $node_b$ becomes a main-initiator because of its higher priority on the node's ID. Each $node_i$ except for the initiators begins the snapshot algorithm and stores its local state when it receives the marker for the first time at $ckpt_{i1}$. The main-initiator terminates the partial snapshot algorithm when it broadcasts termination messages, and each $node_i$, except the main-initiator $node_b$, terminates when it receives the termination message and receives all the markers that the node has to receive. The termination messages include the node list and each $node_i$ has to receive the markers from the nodes in the node list. Then, each node can determine which message links should be recorded, where the messages received between $ckpt_{i1}$ and $ckpt_{i2}$ are stored as in-transit messages. Each node must store the message link state in its checkpoint to achieve consistency.

Next we explain how the corresponding execution (the SSS simulation) of SSS algorithm is constructed from a given CSS algorithm execution. The SSS simulation has the same checkpoints as an original execution of CSS algorithm, while keeping the dependency of the application messages. Figure 7 shows the SSS simulation of the CSS algorithm execution in Fig. 6. We assume that the transfer pattern of the application messages is the same at both the CSS execution and the SSS simulation to make identical system behavior. We show the construction by using this example. In the SSS simulation, a virtual initiator *V-Init* is introduced to the original distributed system. At any time, *V-Init* can send virtual application messages (*dummy messages*) that have no effect on the application (i.e., they are ignored when received). In the SSS simulation, *V-Init* sends two dummy messages to $node_b$ and $node_d$ before initiating the snapshot algorithm to establish a communication-relation. First, we assume that *V-init* initiates SSS algorithm at $ckpt_{v1}$ and sends markers to $node_b$ and $node_d$ because of the communication-relation created by the two dummy messages. In this simulation, markers from *V-init* can be received at the exact same timing with the CSS execution ($ckpt_{i1}$ in Fig. 6) because the message delivery can be delayed arbitrarily in the asynchronous distributed system. Therefore, the initiating time of each node is the same as the CSS execution example. Next, in the CSS execution (Fig. 6), the markers initiated by $node_b$ and $node_d$ are replaced by the markers from *V-Init*. The initiator value contained in them is changed to *V-Init*. Therefore the destinations of the DS information messages are also changed to *V-Init*. The order of the arrivals of the DS information messages at *V-Init* can be decided arbitrarily. We assume that the snapshot group of the CSS algorithm execution of Fig. 6 is the group of all nodes. Accordingly, in the SSS simulation, *V-Init* terminates the partial snapshot algorithm when it receives DS information messages from all the nodes. In Fig. 7, *V-Init* terminates the algorithm at $ckpt_{v2}$ and broadcasts the termination messages to all the other nodes. Finally, each node $node_i$ except *V-Init* terminates the algorithm at $ckpt_{i2}$ when it receives the termination message.

We show that the SSS simulation is a possible execution of SSS algorithm. The markers and the DS messages are clearly enabled at their sending, and don't contradict the event dependencies. In the SSS simulation, each termination message depends on all of the DS messages each of which depends on $ckpt_{i1}$. In the CSS execution of Fig. 6, the termination of the main initiator ($node_b$) depends on all of the DS messages after the communication of the collision-handling messages, and the termination of the nodes other than $node_b$ depends on its termination. On the other hand, the DS information message of each $node_i$ depends on $ckpt_{i1}$. In the CSS execution, each $ckpt_{i2}$ depends on all of $ckpt_{j1}$. Therefore, we assume that each $node_i$ receives the termination message at $ckpt_{i2}$ in the SSS simulation without producing any contradiction on event dependency. As those, the SSS simulation presents one possible execution of the SSS algorithm.

Finally, we show that the two partial snapshots are the same. To ensure that, we show the following three claims: (a) The set of nodes participating in the partial snapshot (snapshot group) by CSS algorithm is the same as that by SSS algorithm. (b) For any node, a local state recorded by CSS is the same as that by SSS. (c) For any node, the state of every message link recorded by CSS is the same as that by SSS. For each $node_i$, $ckpt_{i1}$ in the SSS simulation and $ckpt_{i1}$ in the CSS execution occur at the same time relative to the events of receiving application messages, and then the recorded local states are identical among them. This proves (b). Similarly, all corresponding DS information messages

are the same between the two executions. In the SSS simulation, all the DS messages are received by *V-Init* and the order of receiving them can be decided arbitrarily, because it does not affect the event dependency. Then we can assume that the order is same as the order of receiving DS information messages by main initiator $node_b$ in the CSS execution. As a result, in the SSS simulation, *V-Init* decides the same snapshot group as $node_b$ does in the CSS execution. Then condition (a) is satisfied. Last, $ckpt_{i1}$ and $ckpt_{i2}$ of both executions occur at the same timing between the two executions relative to the events of receiving application messages, and their snapshot groups are identical. Then the recorded message links and their contents are exactly the same between them. This implies condition (c).

4. Performance Evaluation

In this section, we present our experimental results to show the practical performance of CSS algorithm.

We evaluated our proposed algorithm with the following experiments; two kinds of latency experiments and one throughput experiment. In the first latency experiment, we compared the response times among the following three systems; one taking NO checkpoint, another taking checkpoints with SSS algorithm, and the other taking checkpoints with CSS algorithm. To evaluate the overhead of our snapshot algorithm, we made experiments with changing the frequency of sending requests and taking checkpoints. In the second latency experiment, we compared the response times between two systems; one taking checkpoints with SSS algorithm and the other with CSS algorithm. In this experiment, we controlled the collision ratio to check the effect of collisions on the response time. In the throughput experiment, we compared two systems: NO checkpoints and checkpoints with CSS algorithm.

Our experiment was made for two different scales of distributed systems: 12 PCs and 36 PCs to confirm the scalability of our system.

4.1 Experiment Environment

As we mentioned in the introduction, a partial snapshot algorithm works efficiently when it is applied to dynamic and large-scale distributed systems such as web-service [29] systems specified by W3C [11]. For the performance evaluation of our algorithm, we implemented a sample application of market place that is the standard benchmark Testbed for W3C's web-service platform [30].

The scales of experiment environment, 12 PCs and 36 PCs, might seem to be relatively small for a large-scale market place. However, the scales are enough for our evaluation because the communication of market place are done among (dynamically changing) independent small groups of nodes, and then the number of the nodes that joined each partial snapshot is relatively small. Moreover, if we increase the frequency of partial snapshots, the size of the groups becomes smaller. Actually, for those scales, partial snapshots seldom spread to the entire system in a reasonable frequency, for example, each timing that a task of a client is completed.

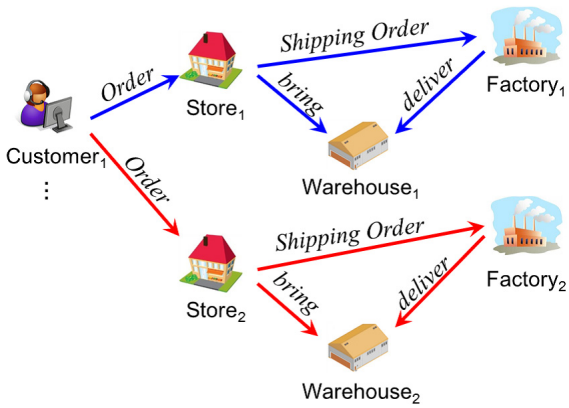

Fig. 8 Structure of implemented web-service.

Table 1 Specifications of PCs.

Number of PCs	12 or 36
CPU	Intel Core i3 2100 (3.1 GHz)
RAM	DDR3 4 GB
HDD	S-ATAII 500 GB
OS	CentOS 5.7
Platform	Apache Tomcat 6, Axis2 1.4.1

For the experiments, we implemented a standard application of a web-service system: a shopping supply chain consisting of several service nodes (servers). In this application, customer nodes send requests to several store nodes. When a store node receives a request from a customer, it begins to process the received request and exchanges message between warehouse and factory nodes if necessary. Note that some nodes do not communicate with warehouse or factory nodes and this implies that the dependencies among nodes might be generated or not. The structure of our application is depicted in Fig. 8.

The system environment is shown in Table 1.

In the following, we show the setting of each experiment and the experimental results.

4.2 Latency by request and checkpoint frequency

In this experiment, snapshot algorithms are initiated by two customers for every fixed number of requests that they have received. We call this interval a *checkpoint interval*. To control the frequency of requests, we set up a new node called a coordinator node. Requests are issued to customers in a round-robin fashion by the coordinator node at every fixed time interval. We call the interval at which requests arrive a customer *request interval*. For the system with SSS algorithm, an initiator can begin a new execution of SSS algorithm after the terminations of the snapshot algorithms initiated by other customers to avoid a collision. In both the systems with SSS and CSS, after its own initiation, the initiator has to wait for the termination before processing the next request to confirm the commitment of the previous requests.

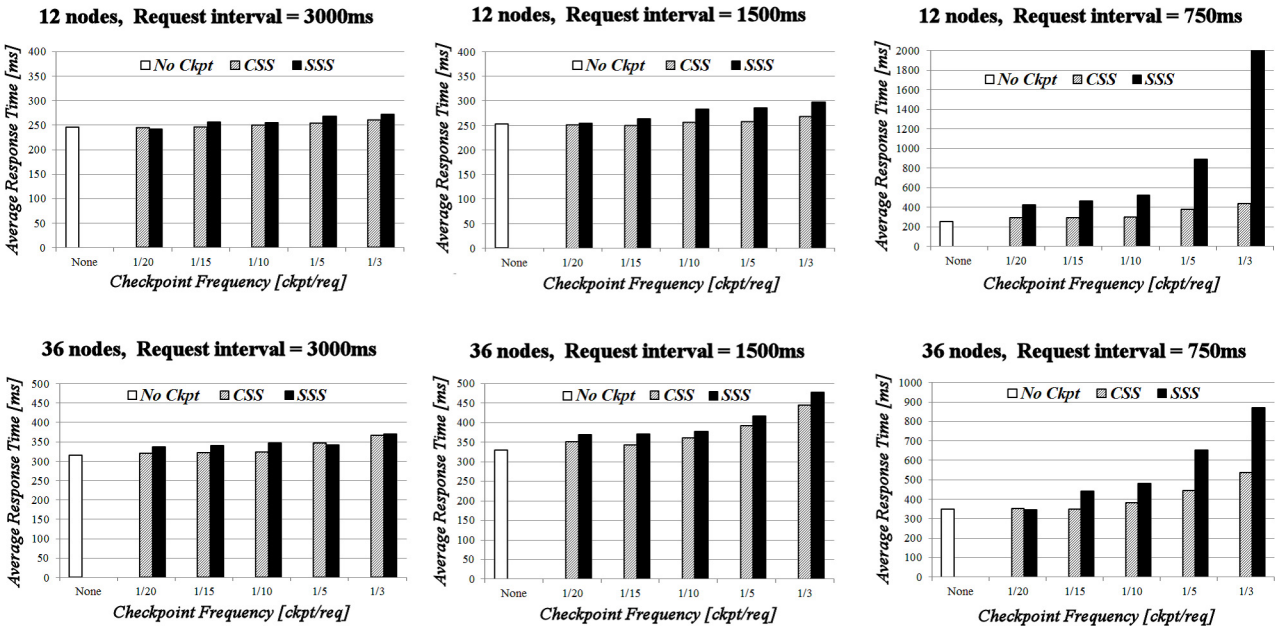

Fig. 9 Result of latency experiments.

In the experiment, we measured the average response times of the requests by changing the frequency of the requests and the checkpoints whose frequency is controlled by the request and checkpoint interval. Of course, the more frequent checkpoints are taken, the less nodes are involved in each checkpoint because the extent of dependency among the nodes through communication has not much grown. After that, we measured the average response times of the requests. The result is shown in Fig. 9.

From this result, the difference of the latencies among the three systems is slight, except when the request and checkpoint intervals are short. This means that the overhead of CSS and SSS algorithms for checkpointing can be ignored compared with the normal operation of applications. In the high frequency (high processing load) case, the collision ratio is increasing and the response of the system with SSS is delayed.

4.3 Latency by collision ratio

The basic setting of the experiment is the same as the first one. We fixed the checkpoint interval to every five requests and the request interval to 900 ms. For the system with CSS algorithm, we call the ratio of the occurring collisions between partial snapshots the *collision ratio*. In this experiment, we changed the collision ratio by manipulating the timing of initiating the checkpoint algorithm for one of the customers. For example, if the coordinator node sends the checkpoint request to an initiator just after the other node initiates the snapshot algorithm, they will collide. On the other hand, if the coordinator node sends the checkpoint request to an initiator after sufficiently long time to complete the checkpoint by the previous initiator, they will not collide. We changed the timing of sending the checkpoint requests to vary the collision ratio by holding specified checkpoint frequencies.

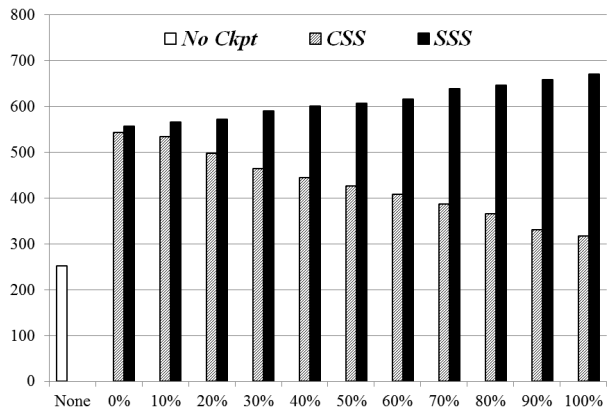

Fig. 10 Average response time by changing collision ratios.

Figure 10 shows the average response times of both systems with CSS and SSS algorithms by changing collision ratios. The response time of SSS algorithm increased, but CSS decreased because the system with SSS algorithm and a high collision ratio has to wait for a long time until the previously scheduled checkpointings, and the processing of subsequent requests are blocked. On the other hand, in the system with CSS, even if a collision occurs, the collided partial snapshots are combined. From a different point of view, the resulting snapshot is divided and processed in parallel. Therefore, we conclude that the advantage of speed up by this parallelization exceeds the merging overhead, and fast response is achieved.

4.4 Throughput

The setting of this experiment was the same as that of la-

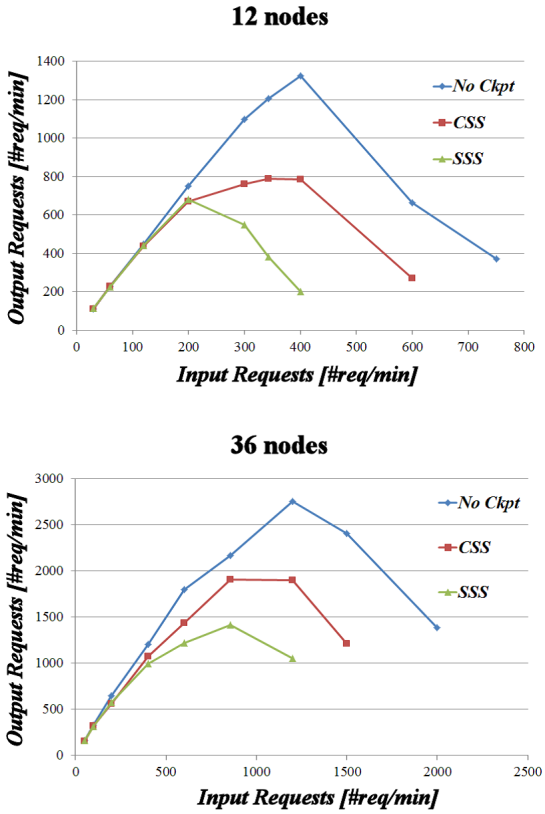

Fig. 11 Result of throughput experiments.

tency. We evaluated the processing capacity by this throughput experiment. The result is shown in Fig. 11. The extra consumption of resource by SSS and CSS algorithms is almost the same. However, after the resource limit, the throughput capacity of the system with SSS algorithm is exhausted faster. Because of the high collision ratio around the limit, requests are blocked and exceeding demands for resource consumption are accumulated.

5. Conclusion

We proposed a concurrent partial snapshot algorithm CSS. This algorithm makes it feasible to take snapshots of large-scale and dynamic distributed systems characterized by the participation of a prodigious number of nodes. CSS algorithm can deal with situations called collisions in which two or more algorithms are simultaneously executing on a single node. Any node in a distributed system can initiate the partial snapshot algorithm. We implemented a virtual web-service and used it to evaluate CSS algorithm. In an environment with frequently overlapping algorithms, CSS algorithm operated efficiently. Therefore, we expect that it can efficiently support the fault-tolerance of large-scale dynamic distributed systems.

Acknowledgements

The authors would like to thank Prof. H. Kakugawa and Prof. F. Ooshita at Osaka University for meaningful discussions about improving this paper. This work is supported in part by Global COE Program and Grant-in-Aid for Scientific Research ((B) 22300009) of JSPS.

References

[1] K. Chandy and L. Lamport, "Distributed snapshots: Determining global states of distributed systems," ACM Trans. Computer Systems, vol.3, no.1, pp.63–75, 1985.
[2] S. Moriya and T. Araragi, "Dynamic snapshot algorithm and partial rollback algoithm for internet agents," Proc. 15th International Symposium on DIStributed Compuiting (DISC 2001) Brief Announcecments, pp.23–28, 2001.
[3] S. Moriya and T. Araragi, "Dynamic snapshot algorithm and partial rollback algoithm for internet agents," IEICE Trans. Inf. & Syst. (Japanese Edition), vol.J86-D-I, no.5, pp.301–317, May 2003.
[4] Y. Kim, et al., "Brief announcement: A concurrent partial snapshot algorithm for large-scale and dynamic distributed systems," Proc. 13th International Symposium on Stabilization, Safety, and Security (SSS 2011), pp.445–446, 2011.
[5] A.S. Tanenbaum and M.V. Steen, "Distributed systems: Principles and paradigms," Second ed., Pearson, 2006.
[6] R. Garg, et al., "Efficient algorithms for global snapshots in large distributed systems," IEEE Trans. Parallel Distrib. Syst., vol.21, no.5, pp.620–630, 2010.
[7] A.D. Kshemkalyani, et al., "Fast and message-efficient global snapshot algorithms for large-scale distributed systems," IEEE Trans. Parallel Distrib. Syst., vol.21, no.9, pp.1281–1289, 2010.
[8] H. Higaki, et al., "Checkpoint and rollback in asynchronous distributed systems," Proc. 16th Annual Joint Conference of the IEEE Computer and Communications Societies, vol.3, pp.990–1005, 1997.
[9] A.D. Kshemkalyani and M. Singhal, Distributed Computing: Principles, Algorithms and Systems, Cambridge University Press, 2008.
[10] Web Services Architecture Requirements, W3C Working Group Note 11 Feb. 2004, http://www.w3.org/TR/wsa-reqs/#id2604831
[11] World Wide Web Consortium, http://www.w3.org/
[12] RosettaNet, http://www.rosettanet.org
[13] R. Koo and S. Toueg, "Checkpointing and roll-back recovery for distributed systems," IEEE Trans. Softw. Eng., vol.13, no.1, pp.23–31, Jan, 1987.
[14] M. Spezialetti and P. Kearns, "Efficient distributed snapshots," Proc. 6th International Conference on Distributed Computing Systems, pp.382–388, 1986.
[15] R. Prakash and M. Singhal, "Maximal global snapshot with concurrent initiators," Proc. 6th IEEE Symposium of Parallel and Distributed Processing, pp.334–351, 1994.
[16] T.T.-Y. Juang and S. Veckatesan, "Crash recovery with little overhead," Proc. 11th International Conference on Distributed Computing Systems, pp.454–461, 1991.
[17] M.J. Fischer, N.D. Griffeth, and N.A. Lynch, "Global states of a distributed system," IEEE Trans. Softw. Eng., vol.SE-8, no.3, pp.198–202, May 1982.
[18] F. Mattern, "Virtual time and global states of distributed systems," Parallel and Distributed Algorithms: proceedings of the International Workshop on Parallel and Distributed Algorithms, pp.215–226, 1989.
[19] F. Mattern, et al., "Efficient algorithm for distributed snapshots and global virtual time approximation," J. Parallel and Distributed Computing, vol.18, no.4, pp.423–434, 1993.
[20] T.H. Lai and T. Yang, "On distributed snapshots," Inf. Process. Lett., vol.25, no.3, pp.153–158, 1987.
[21] R. Garg, V. Garg, and Y. Sabharwal, "Scalable algorithms for global snapshots in distributed systems," Proc. 20th Annual International Conference on Supercomputing, pp.269–277, 2006.

[22] R. Garg, V. Garg, and Y. Sabharwal, "Efficient algorithms for global snapshots in large distributed systems," IEEE Trans. Parallel Distrib. Syst., vol.21, no.5, pp.620–630, 2010.
[23] B. Randell, "System structure for software fault-tolerant," IEEE Trans. Softw. Eng., vol.1, pp.220–232, 1975.
[24] R. Baldoni, F. Quaglia, and B. Ciciani, "A VP-accordant checkpointing protocol preventing useless checkpoints," Proc. International Symposium on Reliable Distributed Systems, pp.61–67, 1998.
[25] D. Briatico, A. Ciuffoletti, and L. Simoncini, "A distributed domino-effect free recovery algorithm," Proc. IEEE International Symposiuim on Reliability Distributed Software and Database, pp.207–215, 1984.
[26] J.M. Helary, A. Mostefaoui, R.H.B. Netzer, and M. Raynal, "Preventing useless checkpoints in distributed computations," Proc. IEEE International Symposium on Reliable Distributed Systems, pp.183–190, 1997.
[27] M. Elnozahy, et al., "A survey of rollback-recovery protocols in message-passing systems," ACM Computing Surveys, vol.34, Issue 3, pp.375–408, 2002.
[28] A. Mostefaoui, et al., "From static distributed systems to dynamic systems," Proc. 24th IEEE Symposium on Reliable Distributed Systems (SRDS), pp.109–118, 2005.
[29] Web Services Architecture, W3C Working Group Note, 2004, http://www.w3.org/TR/ws-arch/
[30] D.F. Carcia, et al., "Benchmarking of web services platforms - An evaluation with the TPC-App benchmark," Proc. International Conference on Web Information Systems ans Technologies (WEBIST), pp.11–13, 2006.

Yonghwan Kim is a Ph.D candidate at Graduate School of Information Science and Technology in Osaka University, in Osaka, Japan. He received the B.E. double-degree in Electronics and Informatics from Soongsil University in Seoul, Korea, in 2009. And he received the M.E. degree in Informatics from Osaka University in 2011. He has lots of developing experience as a member of Samsung Software Membership at Samsung Electronic Company, Korea. The areas of research interests include distributed computing, fault tolerance and distributed file systems.

Tadashi Araragi is a Senior Research Scientist at NTT Communication Science Laboratories, NIPPON TELEGRAPH AND TELEPHONE CORPORATION. He received the B.E. and M.E. degrees in Mathematics from Tokyo University, in 1985 and 1987, respectively, and D.E. degree in Informatics from Kyoto University in 2006. He joined Communications and Information Processing Labs. of NTT, Yokosuka, Japan in 1987. His research interest includes formal verification, analysis of security protocols and fault tolerance of distributed systems. Dr. Araragi is a member of IEICE and IEEE Society.

Junya Nakamura received the B.E. and M.E. degrees from the Department of Knowledge-Based Information Engineering, Toyohashi University of Technology, Japan, in 2006 and 2008 respectively. He is currently a Ph.D student at Graduate School of Information Science and Technology, Osaka University. His research interests include distributed algorithms and dependability of distributed systems, especially Byzantine fault tolerance.

Toshimitsu Masuzawa received the B.E., M.E. and D.E. degrees in computer science from Osaka University in 1982, 1984 and 1987. He had worked at Osaka University during 1987–1994, and was an associate professor of Graduate School of Information Science, Nara Institute of Science and Technology (NAIST) during 1994–2000. He is now a professor of Graduate School of Information Science and Technology, Osaka University. He was also a visiting associate professor of Department of Computer Science, Cornell University between 1993–1994. His research interests include distributed algorithms, parallel algorithms and graph theory. He is a member of ACM, IEEE, IEICE and IPSJ.