# Realization of An Efficient Concurrent Partial Snapshot Algorithm for Large-scale and Dynamic Distributed Systems

Rentaro Watanabe*, Yonghwan Kim†, Fukuhito Ooshita‡, Hirotsugu Kakugawa* and Toshimitsu Masuzawa*
*Osaka University
Graduate School of Infomrmation Science and Technology, Osaka, Japan
Email: r-watanb@ist.osaka-u.ac.jp, kakugawa@ist.osaka-u.ac.jp, masuzawa@ist.osaka-u.ac.jp
†Nagoya Institute of Technology Graduate School of Engineering
Department of Computer Science and Engineering, Aichi, Japan
Email: kim@nitech.ac.jp
‡Nara Institute of Science and Technology
Graduate School of Information Science, Nara, Japan
Email: f-oosita@is.naist.jp

***Abstract*—This paper presents a new algorithm for taking a *p*artial snapshot in a dynamic distributed system. A *g*lobal snapshot is a collection of local states of *a*ll the nodes in the system and plays important roles in many applications such as checkpoint-based rollback recovery. But it becomes impractical and instead the *p*artial snapshot becomes useful when the distributed system becomes large or dynamic. In the partial snapshot, all the nodes do not store their states, but instead, a node stores its state only when it has causal relation to the node initiating the snapshot algorithm.**

**Moriya and Araragi presented an efficient partial snapshot algorithm for a dynamic distributed system where nodes can join to and leave from the system. But they consider only the case where a *s*ingle node initiates the snapshot algorithm. Kim et al. modified the algorithm so that *m*ultiple nodes can concurrently initiate the snapshot algorithm. This paper presents another method to allow the multiple concurrent initiators. Both the algorithms merge the partial snapshot processes initiated by the initiators and finally one of the initiators coordinates the process for taking the partial snapshot. The new algorithm executes the merge more distributedly than the previous one, and succeeds to reduce the number of messages to take the partial snapshot.**

***Keywords*-Distributed system; Fault tolerance; Checkpoint-based rollback recovery; Snapshot algorithm; Partial-snapshot algorithm;**

## I. はじめに

ネットワークを介して複数の計算機が相互に通信しながら協調して動作するシステムを分散システムと言う．分散システムでは一部のノードの故障が他の多くのノードに影響を及ぼすことがある．また近年の分散システムでは大規模化が進んでいるため，システム全体として故障が発生する確率もそれに伴って高くなる．よってシステムの耐故障性がますます重要になってきている．

分散システム上のあるノードで故障が起きた際にシステムの状態を復元する手法にチェックポイントロールバックリカバリー [1][2] がある．これは分散システムに故障が発生した際，正常なうちに記録しておいた安定状態へシステムを復元するものである．ここで，事前に保存された各ノードとチャネル (メッセージリンク) の状態を合わせてチェックポイントと言い，チェックポイントの状態へ復帰することをロールバックと言う．分散システム全体の状態は分散システムに属する全ノードのチェックポイントの集合である．分散システムにおいて複数のノードが同時に動作を実行することは難しく，各ノードは任意のタイミングでチェックポイントを取得する．よってチェックポイントの取り方によってはロールバックを行った際の分散システムの状態に矛盾が生じる可能性がある．この矛盾についてはこの後説明する．各ノードにおけるあるタイミングまでに発生したメッセージの送受信等のイベント集合の，全ノード分和集合をとったものを Cut と呼び，Cut 内のイベント関係について矛盾のないものを Consistent Cut と呼ぶ．Figure1 は分散システムの動作おける Consistent Cut とそうでない Cut の例である．各 $m_n$(n=1,2,3,4,5) で表される矢印はアプリケーションメッセージの伝送を示している．分散システムにおいて，あるノードが送信していないメッセージを他のノードが受信しているとき，このメッセージを orphan message と言う．分散システム全体で orphan message が存在しない記録の状態を “ 矛盾がない”という [3]．Figure1 の Cut1 について，$node_a$ は $m_1$ 送信と $m_3$ 受信のイベントを，$node_b$ は $m_1$ 受信，$m_2$ 受信，$m_3$ 送信，$m_4$ 送信のイベントを，$node_c$ は $m_2$ 受信のイベントをそれぞれ含んでいる．このとき，どのメッセージも orphan message ではないので Cut1 は Consistent Cut である．すなわちこの Cut は矛盾のない大局状態を表している．Cut2 について，Cut1 のイベントに加え $node_b$ は $m_5$ 受信のイベントを，$node_c$ は $m_4$ 受信のイベントを記録している．このとき，$m_5$ は $node_b$ によって受信イベントは記録されているが，送信イベントはどのノードも記録していない．よって $m_5$ は orpahan message であるので Cut2 は Consistent ではない．よってこの Cut は矛盾を含む大局状態を表している．

各ノードが記録したチェックポイントの集合をスナップショット [4][5] と呼び，また各ノードが協調的に動作して

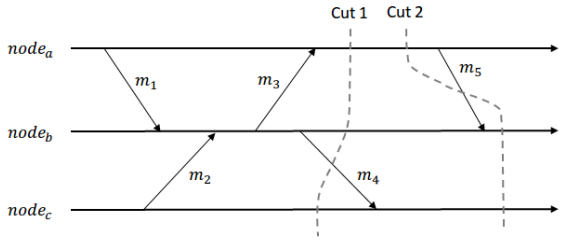


Figure 1. 分散システムにおける Cut の例

矛盾が発生しないようチェックポイントを取りスナップショットを作成するアルゴリズムをスナップショットアルゴリズムと呼ぶ．スナップショットアルゴリズムの 1 つに Chandy と Lamport によるスナップショットアルゴリズム (CL アルゴリズム)[6] がある．CL アルゴリズムは効率の良さと手続きの簡潔さからその代表的なものである．しかし CL アルゴリズムでは各ノードがチェックポイントをとる合図となるメッセージを送信可能なノード全てに送信して伝播させるため，適応するシステムの規模が大きくなると膨大なメッセージ通信が必要となり，実用が難しいという問題があった．

CL アルゴリズムを基に考案された Sub-Snapshot(SSS) アルゴリズム [7][8][9] はこの問題を解決している．これは initiator と動的な因果関係をもつノード集合 (スナップショットグループ) のみでスナップショットをとるアルゴリズムである．これにより同時にスナップショットをとるノード数を削減し，ノードの参加や離脱への対応にも成功したが，SSS アルゴリズムは並行的に実行されることを仮定していないため，大規模なシステムにそのまま適用することは現実的ではない．

SSS アルゴリズムを基に処理の並行実行を可能にした Concurrent Sub-Snapshot(CSS) アルゴリズム [10][11] は，アルゴリズムを並行実行させた場合，スナップショットグループの重複 (衝突) が起きたスナップショットグループ同士を統合し，以降 1 つのスナップショットグループとして機能するよう改良された．しかし CSS アルゴリズムでは，衝突が連鎖的に発生した場合に多くのメッセージ通信を必要とする．

そこで本稿では，SSS アルゴリズムの並行実行における衝突時，CSS アルゴリズムに見られるサブスナップショットグループの統合という処理を行わず，矛盾なくスナップショットを作成する手法を提案する．この手法では統合処理を行わないため，CSS アルゴリズムにおける統合のためのメッセージ伝送が不必要となる．すなわち，ネットワークにおけるノードの連結度 (Connectivity)，スナップショットアルゴリズムが起動される頻度 (Frequency) によって，CSS アルゴリズムと比較すると大まかには以下のような特徴を持つ．

- Connectivity,Frequency が大きい場合，メッセージ複雑度が大幅に改善され，小さくなる．
- そうでない場合でもメッセージ複雑度が大きくなることがない．
- メッセージが運ぶ情報量が減少する．

本稿で紹介する既存研究におけるアルゴリズムや提案アルゴリズムでは，以下のようなモデルを想定している．

- 各ノードはシステムに含まれるノード集合において固有の識別子 (ID) が与えられている．
- アプリケーションレベルで伝送されるメッセージ (アプリケーションメッセージ) を除いたメッセージは ID を知っているノードへ送信可能である．
- 各ノードはネットワークシステムの全容 (総ノード数，ネットワークの構造，自身が送受信したことのないアプリケーションメッセージ) を知らない．
- 各通信リンク上では First In-First Out(FIFO) の制約に則った通信が行われる．

続く第 II 章で既存研究における各アルゴリズムの概要と本研究の目的を説明し，第 III 章で提案アルゴリズムを紹介，プロトコルの解説をする．その後，第 IV 章で提案アルゴリズムの正当性を示し，最後に第 V 章で本稿をまとめと，本研究の今後の課題について述べる．

## II. 既存研究

### A. スナップショットアルゴリズム

*1) CL* アルゴリズム*:* CL アルゴリズム [6] は *marker* と呼ばれる特別なメッセージを用いてシンプルな動作を行うだけで，矛盾のないスナップショットを効率よく作成するアルゴリズムであり，以降で紹介するアルゴリズムの基盤となっている．initiator は最初に局所状態を記録し，各隣接ノードへ *marker* を送信する．initiator 以外のノードは，この *marker* を受信することでアルゴリズムが開始する．初めて *marker* を受信したノードは局所状態を記録し，各隣接ノードへ *marker* を送信する．initiator を含めた全てのノードは，全ての隣接ノードから *marker* を受信するとアルゴリズムを終了する．

CL アルゴリズムの実行に関して，各ノードは隣接ノードを予め知っていなければならないという制約がある．実際の分散システムではノードの参加/離脱などによる動的なネットワーク構造の変化が頻繁に起こるが，CL アルゴリズムは制約によりこれに対応していない．また，各ノードは送信できる全てのノードに *marker* を送信するため，ネットワークが大規模化するとメッセージ数が膨大になる．

### B. 部分スナップショットアルゴリズム

スナップショットアルゴリズムの中でも，ネットワーク上の一部のノードによって構成されたスナップショットを取得するものを部分スナップショットアルゴリズム (Partial Snapshot Algorithm) という．先の第 II-A1 節で紹介した CL アルゴリズムは，initiator と連結する全てのノードをスナップショットに含むため，部分スナップショットアルゴリズムではない．大規模なシステムへの適用を考えた場合，スナップショットの正当性を保った上で最小限のノードやリンクのみにより構成されるスナップショットを取得する方がアルゴリズム実行の際のメッセージ複雑度などのコスト面で有用である．以降では既存の部分スナップ

ショットアルゴリズムについて紹介する．

*1) SSS* アルゴリズム*:* SSS アルゴリズム [7][8][9] は，CL アルゴリズムを基に動的なノードの参加/離脱への対応と *marker* 送信回数の削減に成功している．これはアルゴリズム内で initiator と動的な因果関係を持つノード集合 (スナップショットグループ) を決定し，その集合に含まれるノードのみとスナップショットをとることによる．

このアルゴリズムではスナップショットグループを決定するため，依存関係の概念を導入する．各ノードは自身と依存関係を持つノード集合を管理するため，局所変数 $DS$,$DS^-$ を持ち，送信/受信，生成/被生成イベントが発生した際，関与したノード ID を局所変数として保持する依存関係集合 ($DS$) へ動的に追加していく．ノードはアルゴリズム開始までに生成した依存関係の情報を $DS$ に記録する．アルゴリズムが開始すると $DS$ の情報を $DS^-$ へコピーし，$DS$ に含まれる情報を消去する．以降で生成された因果関係はそれまで通り $DS$ へ記録する．すなわち，$DS$ には前回 SSS アルゴリズムを開始したタイミング (アルゴリズム実行中であれば実行中のアルゴリズムを開始したタイミング) 以降に生まれた因果関係が記録され，$DS^-$ にはアルゴリズム開始時における因果関係が記録される．

アルゴリズムを開始すると，initiator はまず CL アルゴリズムの場合と同様に局所状態を保存し，ID を添付した *marker* を，保持している $DS^-$ に含まれる全てのノードに送信する．初めて *marker* を受信したノードは局所状態を保存し，$DS^-$ に含まれる各ノードに *marker* を送信，さらに initiator に $DS^-$ を送信する．initiator はスナップショットグループを決定するため，受信した $DS^-$ の集合 $MkTo$ とその $DS$ の送信元ノード ID の集合 $MkFrom$ を管理する．$MkTo$ に含まれる集合の和集合と $MkFrom$ が一致したとき，その集合が決定したスナップショットグループである．その後 initiator は，各ノード $node_i$ について，$node_i$ を含む $DS^-$ を initiator へ送信したノードの集合を計算し，$node_i$ へノード ID の集合を送信してアルゴリズムを終了する．initiator からそのノード ID 集合を受信したノードは，集合に属する全てのノードから *marker* を受信するとアルゴリズムを終了する．

次に SSS アルゴリズムにおいて各ノードが保存するメッセージについて説明する．各ノードはスナップショットの正当性を保証するため，自身の局所状態を保存してからアルゴリズムを終了するまでの間に受信したメッセージのうち，必要なメッセージの受信記録をメッセージリンクの状態として保存する．よってそれらのメッセージを一時的に保管するためのキュー MsgQ を保持している．各ノードはアルゴリズム実行中，依存関係を持たないノード ($DS$ と $DS^-$ の和集合に含まれないノード) へアプリケーションメッセージを送信する際は，その直前に送信先へ *marker* を送信する．またアプリケーションメッセージ受信の際は．そのアプリケーションメッセージの送信元ノードから以前に *marker* を受信していれば，MsgQ には保存せず，*marker* を受信したことのないノードからの受信であれば MsgQ に保存する．このように受信アプリケーションメッセージが MsgQ に蓄積されていくが，その中でも自身と同一のスナップショットに含まれないノードからのアプリケーションメッセージは保存する必要がないため，アルゴリズム終了時，*marker* を受信したことのあるノードからのアプリケーションメッセージを MsgQ から取り出し，メッセージリンクの状態として保存する．

SSS アルゴリズムについて，並行実行が制限されている点に注意が必要である．異なる複数のノードを initiator として複数のアルゴリズムが同時実行された場合，$MkTo$ に含まれる集合の和集合，つまりスナップショットグループの重複 (以降，衝突と呼ぶ) が発生すると，スナップショットに矛盾が生じる可能性がある．

*2) CSS* アルゴリズム*:* 複数の initiator により SSS アルゴリズムを並行的に実行できるよう改良したものが CSS アルゴリズム [10][11] である．SSS アルゴリズムの並行実行を可能にするためには，異なる SSS アルゴリズムの実行における衝突が発生した場合でも矛盾がないようスナップショットを取れるアルゴリズムを構築しなくてはならない．CSS アルゴリズムでは，あるグループからの *marker* を既に受信しているノードが別のグループからの *marker* を受信したとき，ノードは衝突を検知する．この衝突したスナップショットグループを矛盾が発生しないようにして統合する．また衝突したスナップショットグループを管理していた initiator のうち一方が統合後のグループを管理し，依存関係などの情報を収集する．この initiator をメイン initiator，他方をサブ initiator と呼ぶ．統合の際，サブ initiator はメイン initiator への論理リンクを生成する．

CSS アルゴリズムではその特性上，メイン initiator 以外のノード (サブ initiator を含む) はどのノードがメイン initiator であるのかを知ることができない．よって，例えばあるノードがアルゴリズムを開始したことで依存関係集合を initiator へ送信した際，その initiator がサブ initiator であればそのメッセージはメイン initiator へ届くまでサブ initiator により伝播されていく．Figure2 に示されるように，衝突が発生したことを initiator へ伝える衝突報告メッセージ，衝突相手の initiator へスナップショットグループの統合を要請する統合要請メッセージについても同様にメイン initiator に届くまで伝播されていく可能性がある．このような性質を考えると，スナップショットグループの衝突が連鎖的かつ大量に発生した場合，メッセージの伝播回数が増加し，アルゴリズム全体としてのメッセージ送信回数 (メッセージ複雑度) は膨大な数になる恐れがある．

## III. 提案アルゴリズム

本章では，SSS アルゴリズムの並行実行中に衝突回数が大きくなった場合，CSS アルゴリズムに比べメッセージ送信回数を削減されるアルゴリズムを提案する．

### *A.* 提案アルゴリズム概要

あるノードが自身の initiator と異なる ID が添付された *marker* を受信したとき，衝突を検知する．このとき提案アルゴリズムではスナップショットグループの統合処理は行わず，既に直接的，あるいは間接的（以降では推移的と表現する）に衝突関係にあるグループの initiator 群から 1

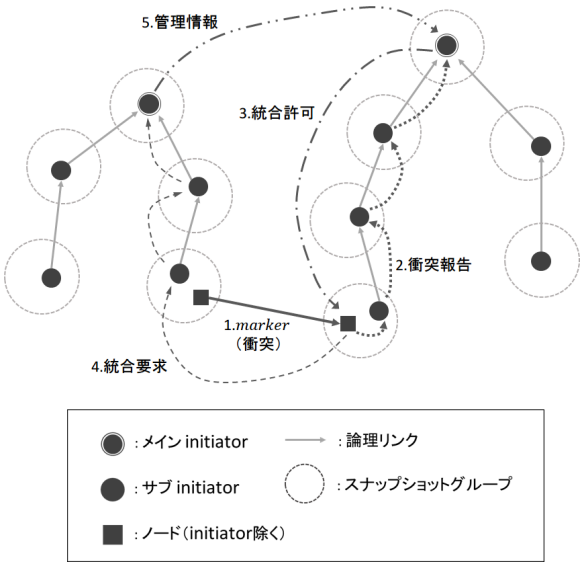


Figure 2. 衝突発生時のメッセージ伝送

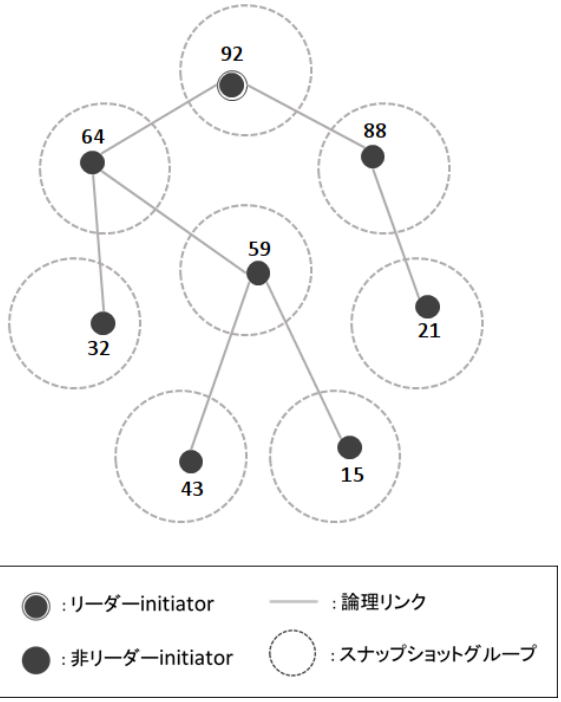


Figure 3. グラフ $T_{init}$ の例

つのリーダーを決定する．このアルゴリズムにおいて，各 initiator はそれぞれ，以前に衝突した際に自身の親となる initiator を決定しており，推移的衝突関係のある initiator 群でグラフ $T_{init}$ を構成している．例を Figure3 に示す．この $T_{init}$ を構成する頂点は initiator であり，これら全ての initiator について，以前に衝突したことのある initiator は全て同一の $T_{init}$ に含まれる．また「任意の隣接している 2 つの initiator について，深さが小さい一方が大きい一方より ID の値が大きい」という特徴を持つ．よって $T_{init}$ の頂点である initiator $init_i$ から深さが大きくなる方へ頂点を辿って行き着く initiator は全て $init_i$ より ID が小さい．同様に深さが小さくなる方へ頂点を辿って行き着く initiator は全て $init_i$ より ID が大きい．このような性質を考えると，グラフ $T_{init}$ には閉路は生まれないため，木である．この木 $T_{init}$ の根にあたる initiator が，この initiator 群のリーダーとなる．STEP1 終了時，各 initiator は管理するスナップショットグループの決定に成功したことをリーダーへ報告し，リーダーが全ての initiator からの報告を確認すると，アルゴリズムの終了を知らせるメッセージを Broadcast する．

本稿では提案アルゴリズムにおいて，各 initiator が自身が管理するスナップショットグループを決定するまでのフェーズを STEP1 と呼ぶ．また各 initiator が正常にスナップショットグループを決定していること，すなわち正常に STEP1 を完了していることをリーダーへ報告し，リーダーが各 initiator へアルゴリズムの終了知らせるまでのフェーズを STEP2 と呼ぶ．以降の第 III-B 節，第 III-C 節でそれぞれのプロトコルを詳しく説明する．

### B. STEP1

以降では，各 initiator が管理するスナップショットの範囲をローカルスナップショットグループ，推移的衝突関係にある initiator 群が管理するローカルスナップショットグループの集合，すなわち，アルゴリズムにより得られる最終的なスナップショットの全範囲をグローバルスナップショットグループと呼ぶ．

STEP1 では，各ノードは第 II-B1 節で紹介した SSS アルゴリズムと同様の振る舞いをしてローカルスナップショットグループを決定する．ただし，並行実行中に衝突が発生した場合はローカルスナップショットグループの統合を行わず，衝突した各 initiator のリーダーのどちらを新たのリーダーに決定する．このとき発生した衝突関係によって $T_{init}$ を更新する．以降で衝突時の動作について説明する．衝突の原因となった *marker* 伝送の受信側ノードを被衝突ノード，送信側のノードを衝突誘起ノードと呼ぶ．

衝突が発生すると，被衝突ノードは自身の initiator へ，衝突の原因となった *marker* に乗っている ID(衝突誘起ノードの initiator の ID) を，*NewInit* メッセージに乗せて送信する．initiator は *NewInit* メッセージによって衝突を知り，被衝突ノードへ *NewInit* メッセージを受信したことを知らせる *Permit* メッセージを送信，更に衝突相手の initiator へ，自身の親 initiator の ID を知らせる *CompId* メッセージを送信する．ただし親 initiator とは，推移的衝突関係にある initiator 群による木において自身の親となっている initiator である．被衝突ノードは *Permit* メッセージを受信すると，自身の ID のみを含めた集合 $DS_c$ を *Collision* メッセージに載せ，衝突誘起ノードの initiator へ送信する．衝突誘起ノードの initiator はこれを受け取ると，通常の *DsInfo* メッセージを受信した場合と同様の処理を行い処理を続ける．

一方，*CompId* メッセージを受信すると，自身の親 initiator の ID とメッセージに添付された ID の値を比較する．

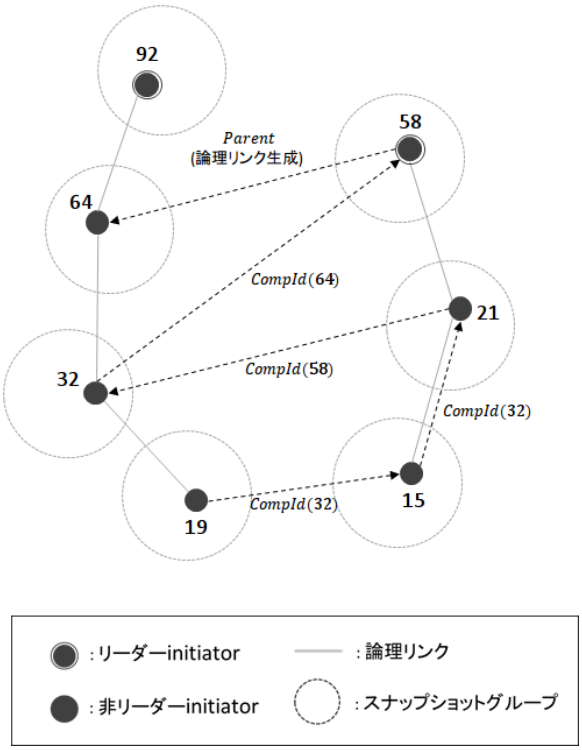


Figure 4. 衝突発生時のメッセージ伝送

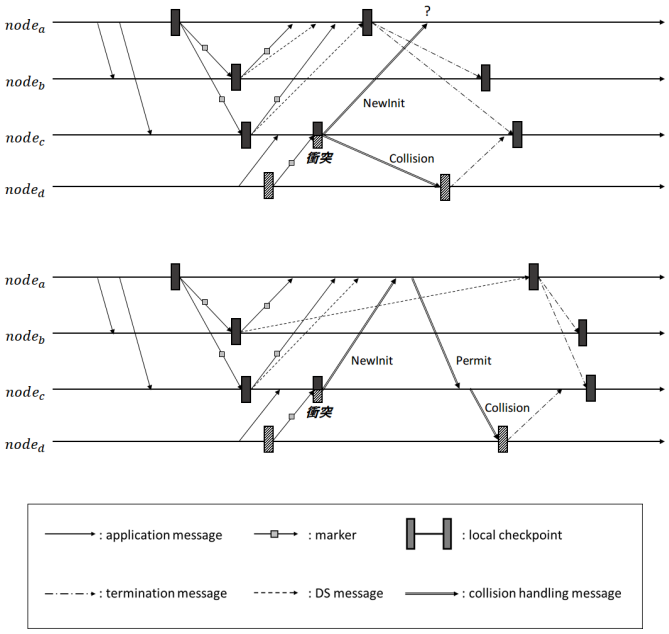


Figure 5. 上:衝突の報告が正常にできない実行例 下:解決策を施した実行例

自身の親 initiator が小さければ，親 initiator へメッセージを転送する．大きければ，メッセージに添付された ID の initiator へ，自身の親の ID を載せた *Collision* メッセージを送信する．このプロセスにより $CompId$ メッセージは，非衝突側と衝突誘起側の属する $T_{init}$ を跨ぎ，根に向かって添付する ID を変更しながら転送され続ける．どちらかの $T_{init}$ の根 initiator まで到達して ID の大小比較を行った結果，自身の ID が小さかった場合，そのとき $CompId$ に添付された ID の initiator の子になるため，$Parent$ メッセージを送信する．このときの $CompId$ メッセージの送信，もしくは $Parent$ メッセージの受信の際に ID を変数に記録しておくことで，各 initiator は $T_{init}$ における自身の親と子の情報を保持する．

Figure4 に衝突発生時のメッセージ伝送イメージを示す．

衝突が発生した場合，提案アルゴリズムにおいて被衝突ノードがすべき処理は本来，(1) 自身の initiator へ衝突誘起ノードの initiator の ID を送信．(2) 衝突誘起ノードの initiator へ $DS_c$ を送信．の 2 つである．しかしこの 2 つの処理のみを逐次的に実行すると，被衝突ノードが衝突を検知し initiator へ *NewInit* メッセージが伝送されるまでの間に以下の処理が共に実行されるケースにおいて，取得するスナップショットに矛盾が生じる可能性がある．

- 被衝突ノードが $DS_c$ を衝突誘起ノードの initiator へ送信
- initiator が STEP1 の処理を終了

このようなケースでは，被衝突ノードは衝突の際の処理を衝突誘起ノードの initiator へ行ったにも関わらず，被衝突ノードの initiator は *NewInit* メッセージの受信によって衝突の存在を把握しないまま終了している．すなわち衝突による initiator 間のリンクが作成されていないため，この状態で各 initiator が STEP2 の処理を行うとスナップショットに矛盾が発生する．Figure5 上部に，スナップショットに矛盾が発生する実行例を示す．この問題を解決するため，提案アルゴリズムにおいて被衝突ノードは，initiator が *NewInit* メッセージを受信したことを *Permit* メッセージの受信により確認をとってから，衝突誘起ノードの initiator へ $DS_c$ を載せた *Collision* メッセージを送信する．Figure5 に，この改善策が施された実行例を示す．

また，Figure 6,7 に提案アルゴリズムにおける STEP1 部分のアルゴリズムの擬似コードを示す．

### C. STEP2

STEP2 では，各 initiator 同士が通信を行い，全ての initiator が正常に STEP1 でスナップショットグループを決定していることを確認する．

まず STEP2 での各 initiator の基本動作を説明する．STEP2 において各 initiator は，STEP1 で作成した木 $T_{init}$ によって通信を行う．前述の通り，$T_{init}$ において各 initiator は $T_{init}$ に含まれる少なくとも 1 つの initiator と STEP1 で衝突している．STEP2 を開始した initiator は，自身が $T_{init}$ において葉ならば，STEP1 を終了したことを知らせる *LocalTerm* メッセージを親 initiator へ送信する．葉でない initiator は *LocalTerm* メッセージを受信すると，その送信元を変数 $Received$ へ追加し，自身の子 initiator 全てからメッセージを受信したかどうか確認する．全ての子 initiator から *LocalTerm* メッセージを受信したとき，親へ *LocalTerm*

```
1: procedure INITIALIZE( )
2:     send (self, <Marker, self>)
3: end procedure

4: procedure ONRECEIVE(<Marker, ?init>, ?sender)
5:     if init = ⊥ then
6:         Store Local State
7:         init ← ?init
8:         RcvMkList ← RcvMkList ∪ {?sender}
9:         send(?init, <DSinfo, DS>)
10:        send(∀node_i ∈ DS, <Marker, ?init>)
11:    else if init = ?init then
12:        RcvMkList ← RcvMkList ∪ {?sender}
13:        if FinFlag then
14:            LocalTermination()
15:        end if
16:    else
17:        MsgQ.enqueue(msg)
18:        send(init, <NewInit, ?init>)
19:    end if
20: end procedure

21: procedure ONRECEIVE(<DSinfo, ?localDS>, ?sender)
22:    AllDS ← (AllDS ∪ ?localDS)
23:    DSSender ← DSSender ∪ {?sender}
24:    LocalTermination()
25: end procedure

26: procedure ONRECEIVE(<Fin, ?DRGinfo>, ?sender)
27:    MustRcv ← DRGinfo
28:    FinFlag ← TRUE
29:    LocalTermination()
30: end procedure

31: procedure ONRECEIVE(<NewInit, ?init>, ?sender)
32:    if ¬ FinFlag then
33:        send(?sender, <Permit>)
34:        send(?sender, <CompId, self>)
35:    end if
36: end procedure

37: procedure ONRECEIVE(<Permit>, ?sender)
38:    MsgQ.Dequeue(msg(<Marker, node_q >, init_q))
39:    DS_c ← { self }
40:    send(init_q, <Collision, DS_c >)
41: end procedure

42: procedure ONRECIEVE(<Collision, ?localDS>, ?sender)
43:    AllDS ← (AllDS ∪ ?localDS)
44:    DSSender ← DSSender ∪ {?sender}
45:    LocalTermination()
46: end procedure
```

Figure 6. 擬似コード [STEP1 前半]

メッセージを送信する．このように木 $T_{init}$ の末端から根のリーダー initiator に向かってメッセージを Convergecast し，リーダーが全ての子からメッセージを受信したとき，アルゴリズムの終了を判断する．最後にリーダーはアルゴリズムの終了を知らせる *Termination* メッセージを木 $T_{init}$ 全体へ Broadcast する．全ての initiator はアルゴリズムの終了する前に自身の管理するローカルスナップショットグループに含まれる全ノードへも *Termination* メッセージを送信する．

Figure8 に提案アルゴリズムにおける STEP2 部分のアルゴリズムの擬似コードを示す．

```
1: procedure ONRECEIVE(<CompId, ?id>, ?sender)
2:     if Parent < ?id then
3:         if Parent != id then
4:             send(Parent, <CompId, ?id>)
5:         else
6:             send(?id, <Parent>)
7:             Parent ← ?id
8:         end if
9:     else if Parent > ?id then
10:        send(?id, <CompId, Parent>)
11:    end if
12: end procedure

13: procedure ONRECEIVE(<Parent>, ?sender)
14:    Children ← Children ∪ {?sender}
15: end procedure

16: procedure LOCALTERMINATION( )
17:    if init=self ∧ !FinFlag then
18:        if AllDS ⊆ DSSender then
19:            send(∀node_i ∈ AllDS, <Fin, {node_j|node_j sent the
    marker to node_i} >)
20:            FinFlag ← TRUE
21:        end if
22:    else if MustRcv ⊆ RcvMkList then
23:        terminate step1
24:    end if
25: end procedure
```

Figure 7. 擬似コード [STEP1 後半]

```
1: procedure ONINITSTEP2( )
2:     if Children = ⊥ then
3:         if Parent = self then
4:             Termination()
5:         else
6:             send(init_i ∈ Connected,<LocalTerm>)
7:         end if
8:     end if
9:     ReceiveListCheck()
10: end procedure

11: procedure ONRECEIVE(<LocalTerm>, ?sender)
12:    Received ← Received ∪ {?sender}
13:    if Step1 を終了済み then
14:        ReceiveListCheck()
15:    end if
16: end procedure

17: procedure RECEIVELISTCHECK( )
18:    if Parent ≠ self then
19:        if Received = Children then
20:            send(init_i ∈ Parent,<LocalTerm>)
21:        end if
22:    else
23:        if Received = Children then
24:            send(∀init_i ∈ Children, <Termination>)
25:        end if
26:    end if
27: end procedure

28: procedure ONRECEIVE(<Termination>, ?sender)
29:    if init = self then
30:        send(∀init_i ∈ Children \{?sender}, <Termination>)
31:        send(∀node_i ∈ DS, <Termination>)
32:    end if
33:    Terminate algorithm
34: end procedure
```

Figure 8. 擬似コード [STEP2]

## IV. 正当性

第 IV-A 節で STEP1，第 IV-B 節で STEP2 の正当性をそれぞれ示すことにより，本スナップショットアルゴリズム全体の正当性が保証されることを示す．

### A. *STEP1* の正当性

SSS アルゴリズムの正当性は [7][8][9] において示されている．よって，どのような状況に対する提案アルゴリズムの動作にもそれぞれ対応する SSS アルゴリズムの動作が存在し，それらの取得するスナップショットが同等であることを示すことで提案アルゴリズムの STEP1 における動作の正当性を証明することができる．
提案アルゴリズムにおける局所変数 $DS$，$DS^-$ は SSS アルゴリズムにおけるものと同一である．$DS$ には前回 SSS アルゴリズムを開始したタイミング (アルゴリズム実行中であれば実行中のアルゴリズムを開始したタイミング) 以降に生まれた因果関係が記録され，$DS^-$ にはアルゴリズム開始時における因果関係が記録される．衝突時における被衝突ノードの対応は，衝突時に衝突誘起ノードの ID を $DS^-$ や $DS$ に含むかどうかによって 4 パターンに場合分けされる．以下の通りである．

case1 衝突誘起ノードの ID が被衝突ノードの $DS^-$ と $DS$ 両方に含まれる
case2 衝突誘起ノードの ID が被衝突ノードの $DS^-$ に含まれ，$DS^-$ には含まれない
case3 衝突誘起ノードの ID が被衝突ノードの $DS$ に含まれ，$DS^-$ には含まれない
case4 衝突誘起ノードの ID が被衝突ノードの $DS^-$ と $DS$ 両方に含まれない

各衝突ケースの例を Figure9 に示す．各衝突ケースにおいて，アプリケーションメッセージ $m_1$ 伝送による因果関係によって $node_a$ の $DS^-$ に $node_b$ の ID が保存され，アプリケーションメッセージ $m_2$ 伝送による因果関係によって $node_a$ の $DS$ に $node_b$ の ID が保存される．

以降で，case1 から case4 のどの場合における提案アルゴリズムの動作によって生成されるスナップショットも，SSS アルゴリズムによって生成されるようなスナップショットであることを示す．このとき提案アルゴリズムのある動作に対応する SSS アルゴリズムの動作について説明する．提案アルゴリズムを動作させた際に衝突が発生したスナップショットグループの initiator へ，仮想のノード V-init がアルゴリズム開始前にアプリケーションメッセージを送信し依存関係を作成している状態で，V-init が SSS アルゴリズムを開始した場合の動作を考える．またその他のメッセージの送信タイミング・遅延時間は提案アルゴリズムにおける動作の対応するメッセージとそれぞれ一致するとする．このような SSS アルゴリズムの動作のうち，提案アルゴリズムを動作させた場合に取得するスナップショットと同一のスナップショットを取得するものが必ず存在することを確認する．取得するスナップショットが同等であることは，以下の 3 点が成り立つことと同値である．

- スナップショットグループに属するノード集合が一致する

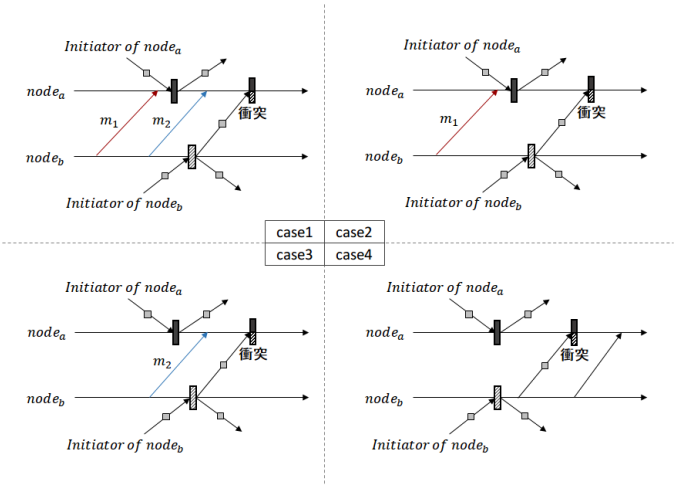


Figure 9. 各衝突ケースの例

- スナップショットグループに属する各ノードについて，保存する局所状態が一致する
- 保存する全てのメッセージリンクの状態が一致する

以降の項で，case1∼4 におけるそれぞれに対する提案アルゴリズムの動作と，その動作に対応する SSS アルゴリズムの動作を示し，それらの取得するスナップショットについて上記 3 点が成り立つことを示す．

*1) case1,case2 :* 衝突発生時の *marker* 送信元ノードの ID が受信ノードの $DS^-$ に含まれており，かつ $DS$ にも含まれている場合 (case1) と含まれていない場合 (case2) をそれぞれ考える．

Figure10 上部に case1 での衝突が，下部に case2 での衝突が発生する提案アルゴリズムの実行例を示す．各 $node_i$ は $ckpt_{i1}$ でアルゴリズムを開始し，$ckpt_{i2}$ でチェックポイントの作成を終了する．$node_a$ と $node_d$ が initiator としてそれぞれ $ckpt_{a1}$,$ckpt_{d1}$ でアルゴリズムを開始する．この 2 つの実行は $node_c$ で衝突し，プロトコルに則った処理が実行される．case1 では衝突発生時，アプリケーションメッセージ $m_1$ によって $node_c$ の $DS^-$ には $node_d$ の ID が記録されており，同様に $m_2$ によって $DS$ にも $node_d$ の ID が記録されている．case2 では衝突発生時，$m_1$ によって $node_c$ の $DS^-$ には $node_d$ の ID が記録されており，また $DS$ には含まれていない．これらの実行は $m_2$ があるかどうかを除いて全く同一の実行である．case1，case2 において $node_d$ からの衝突の際 $node_c$ は，本来の initiator である $node_a$ へ *NewInit* メッセージを送信する．そのメッセージを受信した $node_a$ が送った *Permit* メッセージを受信すると，被衝突ノードは自身の ID のみを含んだ $DS_c$ を載せ，衝突誘起ノードの initiator である $node_d$ へ *Collision* メッセージを送信する．図のケースでは $node_d$ が衝突誘起ノードでもあり，その initiator でもある．

このように衝突を処理し，initiator を除く全ノードは initiator からの終了メッセージを受信するとアルゴリズムを終了する．

このような提案アルゴリズムの case1 における動作に対する SSS アルゴリズムの動作を考える．Figure11 に SSS アルゴリズムの動作例を示す．Figure10 において initiator である $node_a$ と $node_d$ の initiator となるノードとして，新たに V-init と呼ぶ仮想的な initiator を追加する．$node_a$ と $node_d$ の initiator となるため，V-init はアルゴリズムを開始する前に $node_a$ と $node_d$ へアプリケーションレベルの動作に影響を与えない仮想のアプリケーションメッセージを送信し依存関係が成り立っているものとする．またその他のメッセージの送信タイミング・遅延時間は前述した提案アルゴリズムの動作における対応するメッセージとそれぞれ等しいとする．

V-init がアルゴリズムを開始すると，依存関係のある $node_a$ と $node_d$ へ *marker* を送信する．これらのノードは *marker* を受信するとアルゴリズムを開始し局所状態を保存するが，このタイミングは提案アルゴリズムでこれらのノードが initiator としてアルゴリズムを開始し局所状態を保存したタイミングと，アプリケーションメッセージの受信タイミングと比較して一致するものとする．同様に他のノードのアルゴリズム開始タイミングも提案アルゴリズムの動作と一致するものとする．また V-init によりアルゴリズムが開始し *marker* が伝播されるため，*marker* に乗せる initiator の ID も V-init の ID となる．よって各ノードが送信する DS は V-init へ送信される．スナップショットグループが確定すると，V-init はスナップショットグループに含まれる全てのノードへ終了メッセージを送信してアルゴリズムを終了する．V-init 以外のノードは，その終了メッセージを受信するとアルゴリズムを終了する．このような SSS アルゴリズムの動作は確かに起こりうるものである．

このように 2 つのアルゴリズムによって取得されるスナップショットが同等であることを確認する．はじめに，両アルゴリズムの動作において対応する DS メッセージは一致する．SSS アルゴリズムの実行で V-init が DS メッセージの受信により得る情報は，提案アルゴリズムの実行において initiator である $node_a$ と $node_d$ が DS メッセージの受信により得る情報の和である．よって V-init が判断するスナップショットグループは $node_a$ と $node_d$ それぞれが判断するスナップショットグループの和と等しい．すなわち両アルゴリズムの動作におけるスナップショットグループに含まれるノード集合は一致する．次に，両アルゴリズムの動作において対応するそれぞれのノードが局所状態を保存するタイミングは等しいため，各ノードが保存する局所状態についてはそれぞれ一致する．最後に，両実行における各ノードの，アプリケーションメッセージ受信のタイミングと比較した $ckpt_{i1}$・$ckpt_{i2}$ のタイミングは両実行について一致し，かつ両実行においてスナップショットグループは一致するため，保存されるメッセージリンクとその内容は同一である．以上より，提案アルゴリズム STEP1 部分の case1 における正当性は示された．同様に，case2 における提案アルゴリズムの動作に対する SSS アルゴリズムの動作を考えると，それらの取得するスナップショットは case1 と同様の議論によって一致する．よって提案アルゴリズム STEP1 部分の case2 における正当性は示された．

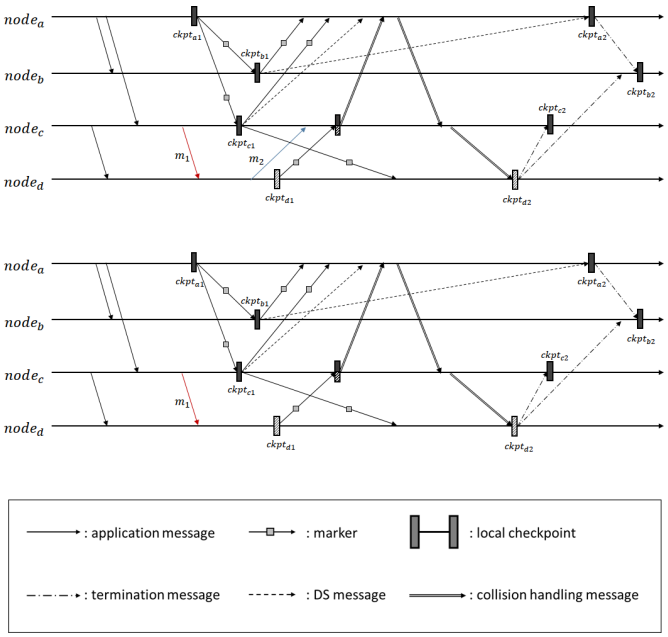


Figure 10. 衝突が発生する提案アルゴリズムの実行例　上：case1 下：case2

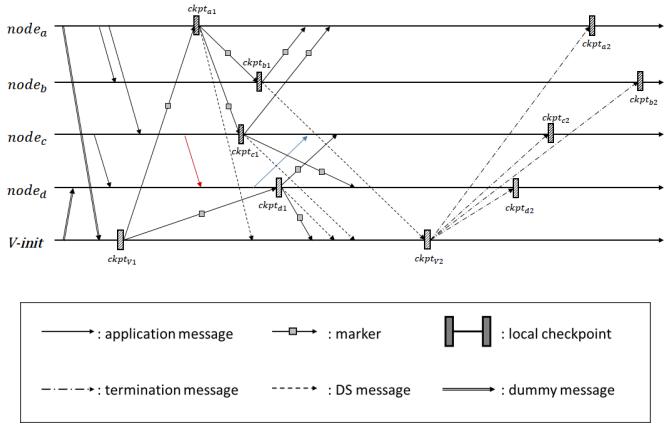


Figure 11. case1 における提案アルゴリズムの動作に対する SSS アルゴリズム実行例

*2) case3,case4 :* 衝突発生時の *marker* 送信元ノードの ID が受信ノードの $DS^-$ に含まれておらず，かつ $DS$ に含まれている場合 (case3) と含まれていない場合 (case4) をそれぞれ考える．

Figure12 上部に case3 での衝突が，下部に case4 での衝突が発生する提案アルゴリズムの実行例を示す．case1,2 の動作例と同様，各 $node_i$ は $ckpt_{i1}$ でアルゴリズムを開始し，$ckpt_{i2}$ でチェックポイントの作成を終了する．case3 では $node_a$ と $node_d$ が $ckpt_{a1}$ と $ckpt_{d1}$ で，case4 では $node_a$ と $node_e$ が $ckpt_{a1}$ と $ckpt_{e1}$ で initiator としてそれ

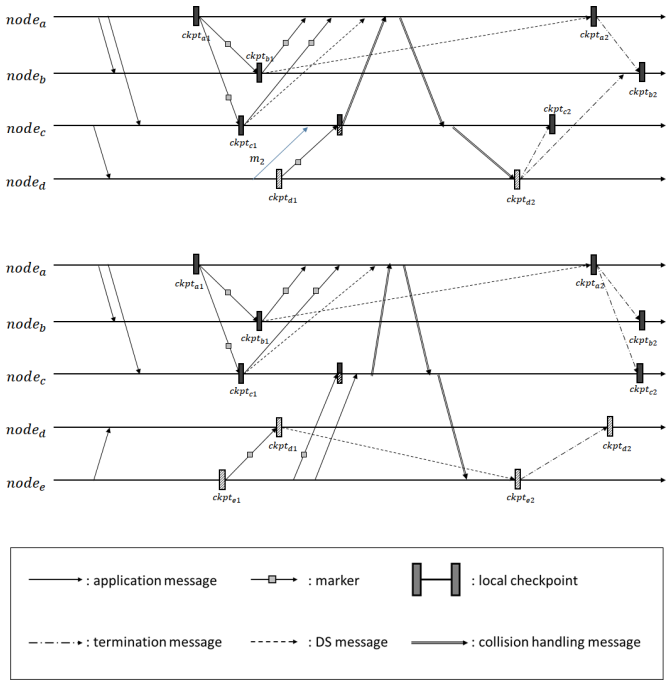


Figure 12. 衝突が発生する提案アルゴリズムの実行例　上：case3 下：case4

ぞれアルゴリズムを開始する．この 2 つの実行は $node_c$ で衝突し，プロトコルに則った処理が実行される．case3 では衝突発生時，$node_c$ の $DS^-$ には $node_d$ の ID が記録されておらず，$m_2$ によって $DS$ には $node_d$ の ID が記録されている．case4 では衝突発生時，$node_c$ の $DS^-$，$DS$ のどちらにも $node_e$ の ID は記録されていない．

case3,case4 での衝突が発生する実行例はどちらも，前項の case1,2 での衝突時における被衝突ノードの処理と同様である．よって前項と同様の議論によって，case3 と case4 における提案アルゴリズムの動作により生成されるスナップショットと対応する SSS アルゴリズムにより生成されるスナップショットは同一であることが言え，case3 と case4 における提案アルゴリズムの動作の正当性は保証される．

よって以下が成り立つ．衝突発生時以外の各ノードの振る舞いは SSS アルゴリズムと全く同一のものであり，かつ，第 IV-A1 項，第 IV-A2 項の証明により，衝突発生時に提案アルゴリズムに則ったどのような処理がなされても，対応する（最終的に同一のスナップショットを取得する）SSS アルゴリズムの実行が存在する．すなわち提案アルゴリズムにおける STEP1 部分の正当性は示された．

### *B. STEP2* の正当性

STEP2 では，第 III-C 節で紹介した通り，STEP1 の動作を終了した各 initiator が，STEP1 実行中に決定したリーダー initiator へ STEP1 を終了したことを報告し，全ての initiator からの報告を確認したリーダーはアルゴリズム終了を許可するメッセージを全ノードへ伝播させる．従って，任意の initiator がアルゴリズムを終了したとき，少なくとも他の全ての initiator は STEP1 を終了した状態であることを証明する必要がある．

本節では，以下の補題を証明することで，STEP2 の正当性を述べる．

定理 *1:* ある initiator $init_i$ が STEP2 の動作を終了したとき，グラフ $T_{init}$ において $init_i$ と連結している (STEP1 において推移的な衝突関係が生まれた) 全ての initiator は必ず STEP1 を終了している．

**定理 1 の証明**　背理法で証明を行うため，以下を仮定する．グラフ $T_{init}$ において，ある initiator $init_i$ が STEP2 を終了するとき，STEP1 を実行中である initiator $init_j$ $(j \neq i)$ が存在する．

第 III-B 節で説明した通り，STEP1 の衝突時のプロトコルによりグラフ $T_{init}$ は木である（閉路を含まない）．$init_i$ が STEP2 を終了するとき $init_i$ は木 $T_{init}$ の根であるリーダー initiator が Broadcast した $Termination$ メッセージを受信済みである．よってリーダー initiator は $T_{init}$ における全ての子 initiator から $LocalTerm$ メッセージを受信済みである．しかしこのとき，仮定により $init_j$ は STEP1 を実行中であるため，親へ $LocalTerm$ メッセージを送信していない．よって $init_j$ の先祖となる initiator は全て $LocalTerm$ メッセージを親へ送信しておらず，リーダー initiator の子で $init_j$ を子孫に持つ initiator も $LocalTerm$ メッセージをリーダー initiator へ送信していないはずである．これはリーダーが全ての子からメッセージを受信済みであることに矛盾するため，定理は示された．
すなわち STEP2 の正当性は示された．■

## V. まとめ・今後の課題

本報告では，依存関係を持つノード間でスナップショットを作成する部分スナップショットアルゴリズムの 1 つであり，並行実行も可能である CSS アルゴリズムを紹介し，CSS アルゴリズムに比べメッセージ複雑度を改善する部分スナップショットアルゴリズムを提案した．また提案したアルゴリズムのプロトコルを 2 つの STEP に分割し，それぞれの正当性を示すことで提案アルゴリズム全体の正当性を示した．今後の課題としては，シミュレーション実験等による CSS アルゴリズムとのメッセージ複雑度の比較，またそのデータに基づく提案アルゴリズムの詳細な評価が挙げられる．

## 謝辞

本研究の全過程における適切な御指導と御協力を賜りました金鎔煥助教授（名古屋工業大学情報工学専攻）に誠意を表わすとともに厚く御礼申し上げます．また本研究を進めるにあたり，素晴らしい研究環境を与えて頂くとともに，多大なるご指導ご助言を賜りました増澤利光教授，角川裕次准教授（大阪大学大学院情報科学研究科）に深く感謝致します．有益なご指導とご協力を頂きました大下福仁准教授（奈良先端科学技術大学院大学情報科学研究科）に感謝の意を表します．そして，日頃から有益な御討論を頂きました増澤研究室の皆様に感謝致します．

## REFERENCES


[1] R Koo and S. Toueg. Checkpointing and roll-back recovery for distributed systems. In *IEEE Transactions on Software Engineering*, Vol. 13(1), pp. 23–31, jan 1987.

[2] Tony T-Y, Juang, and S. Veckatesan. Crash recovery with little overhead. In *Proc. 11th International Conference on Distributed Computing Systems*, pp. 454–461, 1991.

[3] F. Mattern. Virtual time and global states of distributed systems. In *Parallel and Distributed Algorithms*, pp. 215–226, 1989.

[4] Ajay D. K. et al. Fast and message-efficient global snapshot algorithms for large-scale distributed systems,. In *IEEE Transactions on Parallel and Distributed Systems*, Vol. 21-9, pp. 1281–1289, 2010.

[5] R. H. Netzer and J. Xu. Necessary and sufficient conditions for consistent global snapshots,. In *IEEE Transactions on Parallel and Distributed Systems*, Vol. 6(2), pp. 165–169, 1995.

[6] K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. In *ACM Trans. Comput. Syst.*, Vol. 3, pp. 63–75, New York, NY, USA, February 1985. ACM.

[7] 守屋宣, 櫟粛之. インターネットエージェントのための動的スナップショットアルゴリズムと部分ロールバックアルゴリズム. 電子情報通信学会技術研究報告. COMP, コンピュテーション, 第 101 巻, pp. 17–24. 一般社団法人電子情報通信学会, may 2001.

[8] Sen Moriya and Tadashi Araragi. Dynamic snapshot algorithm and partial rollback algoithm for internet agents. In *DISC 2001 Brief Announcecments*, pp. 23–28, 2001.

[9] Sen MORIYA and Tadashi ARARAGI. Dynamic snapshot algorithm and partial rollback algoithm for internet agents. In *IEICE Transactions on Information and Systems*, Vol. J86-D-I, pp. 301–317, 2003.

[10] Yonghwan Kim, Tadashi Araragi, Junya Nakamura, and Toshimitsu Masuzawa. Brief announcement: A concurrent partial snapshot algorithm for large-scale and dynamic distributed systems. In Xavier Dfago, Franck Petit, and Vincent Villain, editors, *SSS*, Vol. 6976 of *Lecture Notes in Computer Science*, pp. 445–446. Springer, 2011.

[11] Yonghwan Kim, Tadashi Araragi, Junya Nakamura, and Toshimitsu Masuzawa. A concurrent partial snapshot algorithm for large-scale and dynamic distributed systems,. In *IEICE Transactions on Information and Systems*, Vol. E97-D, pp. 65–76, 2014.