Vol.2010-DPS-144 No.1
2010/9/17

構造化オーバーレイネットワークに適した分散双方向連結リスト DDLL

安 倍 広 多[†1] 吉 田 幹[†2]

分散双方向連結リスト（双方向リング）は構造化オーバーレイネットワークでしばしば用いられる基本的な分散データ構造の一つである．本稿では，分散双方向連結リストを構築・維持するための自律分散アルゴリズム DDLL を提案する．DDLL では，複数のノードが並行して挿入・削除する場合でも連結リストの構造は常に維持される．また，分散排他制御を用いないため，ノード故障からの修復が容易である．本稿では障害回復を含むアルゴリズムの詳細と形式的な記述を示す．

DDLL: A Distributed Doubly-Linked List Suitable for Structured Overlay Networks

Kota Abe[†1] and Mikio Yoshida [†2]

A distributed doubly-linked list (or bidirectional ring) is a fundamental distributed data structure often used in structured overlay networks. In this paper, we propose a novel decentralized algorithm, called DDLL, for constructing and maintaining distributed doubly-linked lists. DDLL consistently maintains the structure of a linked list even if multiple nodes are simultaneously inserted or deleted. In addition, because DDLL does not require distributed mutual exclusion, it can easily recover the data structure when a node fails. In this paper, we give a detailed explanation and formal description of the algorithm.

†1 大阪市立大学大学院創造都市研究科
Graduate School for Creative Cities, Osaka City University
†2 (株)BBR
BBR Inc.

1. はじめに

構造化オーバーレイネットワークで用いられるデータ構造の一つに，各ノードが双方向の連結リストで接続された分散双方向連結リスト（双方向リング）がある．分散双方向連結リストは，Chord[1], Chord#[2], Skip graph[3] など，多くのアルゴリズムで利用されている．

一般に，オーバーレイネットワークではノードは任意のタイミングで自分自身を挿入・削除する．また，ノードは削除手続きを実行することなく離脱あるいは故障する場合がある．オーバーレイネットワークで用いる分散双方向連結リストでは，これらの点に注意する必要がある．

分散双方向連結リストを構築するための従来の手法は，大きく分けて楽観的アプローチと排他制御アプローチに分類できる．楽観的アプローチ（例えば文献 1)，4) など）では，ノードの挿入・削除に伴って連結リストの構造が一時的に壊れることを許す（この状態はスタビライズ処理によって回復する）．この方式には，障害からの回復が容易という利点があるものの，リスト上で一時的に到達できないノードが存在するという問題がある（障害がない場合でも）．

一方の排他制御アプローチ（例えば文献 5) など）では，ノードの挿入・削除中は，分散排他制御を用いてリストの一貫性を維持する．このため，挿入したノードへは必ず到達できる．この方式でノードを挿入する場合，挿入ノードはまず排他制御を提供するロックノード（典型的には隣接ノード）にロック獲得要求を送ってロックを獲得し，次に隣接ノードのリンクを変更，最後にロックノードにロック解放要求を送ってロックを解放するという流れで処理を実行する．しかし，この方式は障害からの回復が難しい．挿入ノードがロック解放要求を送信する前に故障した場合，ロックが解放されずに残ってしまう．タイムアウトによってロックを強制的に解放する方法も考えられるが，ロック解放要求が単に遅延しているだけという可能性があるため，この方法は本質的に安全ではない．また，タイムアウトを待つ間，別のノードの挿入や削除がブロックされるという問題もある．

本稿では，分散双方向連結リストを構築・維持するための新しい自律分散アルゴリズム DDLL (Distributed Doubly-Linked List) を提案する．DDLL は，これら 2 つのアプローチの問題点を解決する．DDLL では，ノードの挿入・削除時にも連結リストのデータ構造は常に維持されるため，挿入されているノードには必ず到達できる．また，分散排他制御は用いないため，上述したような問題は発生せず，ノード故障から容易に回復できる．著者らの知る限り，既存のアルゴリズムでこの 2 つの特徴を備えたものは存在しない．

© 2010 Information Processing Society of Japan

以下，次の章で DDLL アルゴリズムの前提と記法について簡単に述べ，3 章では故障を考慮しないアルゴリズムを，4 章では故障を考慮したアルゴリズムを述べる．最後に 5 章でまとめと今後の課題について述べる．

2. 準　　備

DDLL は，各ノードが保持する値（キー）によってソートされた分散双方向連結リストを構成する．各ノードは右リンクと左リンクによって他のノードまたは自分自身と接続する．双方向連結リストは 2 つの片方向連結リストに分解できる．それぞれ**右方向連結リスト**，**左方向連結リスト**と呼ぶことにする．本稿では右方向がキーの増加する方向とし，左方向が減少する方向とする．ただし，最大キーを持つノードの右ノードは最小キーを持つノードであり，また最小キーを持つノードの左ノードは最大キーを持つノードである（循環リスト）．

キーは全順序集合の要素とする．また重複するキーが登録されないように，キーの後ろに事実上互いに衝突しない十分なビット数の乱数を付与する．

キー a, b, c に対し，$\mathrm{ord}(a, b, c) = (a \leq b \leq c) \vee (b \leq c \leq a) \vee (c \leq a \leq b)$ と定義する．循環リスト上でキー a から右方向に進んで b，c の順に辿るとき，$\mathrm{ord}(a, b, c) = \mathrm{true}$ である．また，相異なるキー $u, v1, v2$ に対し $\mathrm{ord}(u, v1, v2) = \mathrm{true}$ ならば，キー u からみて $v1$ のほうが $v2$ よりも**右方向で近い**（あるいは $v2$ のほうが $v1$ よりも**左方向で近い**）と表す．

各ノードは同一のアルゴリズムを非同期的に実行する．また，ノード間の通信には信頼性のある非同期な通信路が利用できるものとする．つまり，(ノード故障がない限り）送信したメッセージはいずれ到着するが，到着時間の上限は存在しない．また，メッセージは必ずしも送信順に到着するとは限らない．

ノード u が保持するキーも同じく u で表す．各ノードはいくつかの変数を保持する．主要な変数は次の通りである: **s**（ノードの状態），**l**（左リンク），**r**（右リンク），$\mathbf{l}_{\mathrm{num}}$（左リンク番号），$\mathbf{r}_{\mathrm{num}}$（右リンク番号），**ref**（左リンクによる被参照数）．本稿では記述を平易にするため，l と r はノードの位置 (IP アドレスなど) と，ノードのキーの両方を表すものとして扱う（実際の実装では位置とキーは別々に保持する必要がある）．ノード u の変数 x を，$u.x$ のように表記する．

3. 故障を考慮しないアルゴリズム

この章では，ノード故障やネットワーク故障が発生しない環境における DDLL アルゴリズムを述べる．双方向連結リストにおいてノードを挿入・削除するためには，当該ノードの両側の隣接ノードの右リンクと左リンクを更新する必要があるが，DDLL では，挿入・削除のどちらの場合でもまず左ノードの右リンクを更新し，次に右ノードの左リンクを更新する．

3.1 右リンクの更新方法

まず，左ノードの右リンク更新方法を述べる．ここでは，連結リスト上の隣接ノード a, b 間にノード u を挿入する場合を考える ($a.r = b$, $\mathrm{ord}(a, u, b) = \mathrm{true}$)．なお，$a, b$ を見つける方法は 3.5 節で議論する．

以下のシンプルな方法により，分散排他制御を行わずに a の右リンクを安全に更新できる．

まず，u は右リンクを b に設定する ($u.r := b$)．次に，u は a に対して右リンク更新要求メッセージ（SetR メッセージ）を送信する．SetR メッセージの引数には，新しい右リンク先 $\mathrm{r_{new}}$（この場合は u）に加え，a の現在の右リンク先 $\mathrm{r_{cur}}$（この場合 b）も含めておく．SetR メッセージを受信したノード a は，自身が削除中ではなく，かつ，a の現在の右リンクと $\mathrm{r_{cur}}$ が一致しているときのみ a の右リンクを $\mathrm{r_{new}}$ に更新して確認応答（SetRAck メッセージ）を返す．一致しない場合は拒絶し，否定応答（SetRNak メッセージ）を返す．u が SetRNak を受信した場合，再度隣接ノード a, b を求め，最初からやりなおす[⋆1]．

複数のノードが並行して a, b 間に挿入しようとする場合 ($\mathrm{r_{cur}} = b$)，a は最初に受信する SetR メッセージで a の右リンクを変更するため，引き続く SetR メッセージは拒絶される．また，a が u からの SetR メッセージを受け付けて右リンクを b から u に変更した時点で，u の右リンクは b になっているため，右方向連結リストが途切れることはない．

3.2 左リンクの更新方法

次に，右ノードの左リンク更新処理について考える．

この処理を，右リンク更新処理と対称な方法で行うこともできる．すなわち，左リンク更新要求メッセージ (SetL メッセージ) に，SetR メッセージと同様の引数 (新しい左リンク先 $\mathrm{l_{new}}$ と，現在の左リンク先 $\mathrm{l_{cur}}$) を含めて右ノードに送信する方法である．しかし，この方法は複数のノードが同時に挿入・削除される状況 (churn) で効率が悪い．

例えば，隣接ノード n_0, n_4 の間にノード n_1, n_2, n_3 がこの順に相次いで挿入される場合を考える ($n_i < n_{i+1}$)．ノード $n_i (1 \leq i \leq 3)$ は n_{i-1} に SetR メッセージを送信し，SetRAck メッセージを受信した後，n_4 に SetL メッセージ m_i を送信する (m_i の $\mathrm{l_{cur}}$=n_{i-1}, $\mathrm{l_{new}}$=n_i)．

⋆1 この処理は SetRNak メッセージに a が SetRNak を返した理由と $a.r$ を格納しておくことにより効率化できる．例えば，a の右に別のノード x が先に挿入されていた場合，u は SetRNak メッセージで x の存在を把握できるため，次は a と x（あるいは x と b）の間に挿入すればよい．

© 2010 Information Processing Society of Japan

n_4 は 3 つの SetL メッセージを受信するが，n_4 は m_1, m_2, m_3 の順にしか SetL メッセージを処理できない（これ以外の順序で受信した場合は左リンクの値と $\mathrm{l_{cur}}$ の値が一致せず，SetLNak が返される）．また，ノード故障が発生する場合（いまのところノード故障はない前提だが，後でノード故障がある場合を考える），SetL メッセージの 1 つが届かない可能性がある．このとき，届かない SetL メッセージに依存する他の SetL メッセージまで処理できなくなり，左リンク更新処理が停止してしまう．そこで，次の方式を考案した．

一般に，異なる複数のノードが同一のノードに SetL メッセージを送信するが，以下に述べる方法により，同一のノードを宛先とする SetL メッセージに対して SetL メッセージを送信した順番を表すシーケンス番号を付与できる．この番号を用いることで，左リンク更新処理を SetL メッセージの受信順序に関わりなく実行できる．以下詳細を述べる．

各ノードは，自ノードの SetL シーケンス番号を示す**左リンク番号**と，右ノードの SetL シーケンス番号を示す**右リンク番号**を保持する．ノード u の左リンク番号，右リンク番号を，それぞれ $u.\mathrm{l_{num}}, u.\mathrm{r_{num}}$ で表す．ノード u を挿入した直後は，u の左リンク番号は 0 である．また，隣接ノード a, b が定常状態（a，b に転送中のメッセージが存在しない状態）にある場合，$a.\mathrm{r_{num}} = b.\mathrm{l_{num}}$ である．

SetL メッセージの引数 $\mathrm{l_{newnum}}$ がシーケンス番号を表す．ノードの挿入・削除を以下のように行うことで，同一ノードに送信される SetL メッセージの $\mathrm{l_{newnum}}$ を送信時点でシーケンシャルに増加させることができる（詳細な方法は 3.4 節で述べる）．

ノード u を挿入する場合　ノード a, b 間に挿入するものとする．(1) u は $u.\mathrm{l_{num}}$ を 0 としてから SetR メッセージを a に送信する．SetR メッセージには a の新しい右リンク番号 $\mathrm{r_{newnum}}$（ここでは 0）を含める．(2) SetR メッセージを受信した a は右リンク番号を $\mathrm{r_{newnum}}$ に変更し，SetRAck メッセージを使って変更前の右リンク番号（$\mathrm{r_{prevnum}}$ とする）を u に通知する．(3) SetRAck メッセージを受信した u は $u.\mathrm{r_{num}}$ を $\mathrm{r_{prevnum}} + 1$ に設定し，SetL($\mathrm{l_{new}} = u, \mathrm{l_{newnum}} = u.\mathrm{r_{num}}$) メッセージを b に送信する．

ノード u を削除する場合　$u.l = a$，$u.r = b$ とする．(1) u は SetR メッセージ（引数 $\mathrm{r_{newnum}} = u.\mathrm{r_{num}} + 1$）を a に送信する．(2) SetR メッセージを受信した a は右リンク番号を $\mathrm{r_{newnum}}$ に変更し，u に SetRAck メッセージを返す．(3) SetRAck メッセージを受信した u は，SetL メッセージ（$\mathrm{l_{new}} = a$，$\mathrm{l_{newnum}} = u.\mathrm{r_{num}} + 1$）を b に送信する．

SetL メッセージを受信したノード b は，メッセージの $\mathrm{l_{newnum}}$ と自身の左リンク番号 $b.\mathrm{l_{num}}$ を比較し，前者が大きければ左リンクを $\mathrm{l_{new}}$ に，左リンク番号を $\mathrm{l_{newnum}}$ にそれぞれ更新する．SetL メッセージは必ずしも送信した順番に到着するとは限らないが，このようにすることで，遅れて到着した古い SetL メッセージを無視できる．つまり，SetL メッセージはどのような順序で到着してもよい．

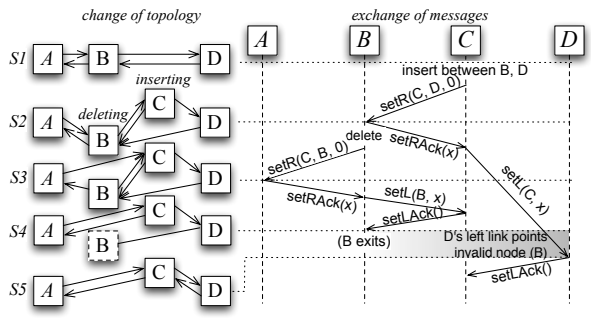

図 1　左ノードが削除済みのノードを指す例

しかし，この方式はタイミングによっては一時的に左リンクが削除済みのノードを指すという問題がある．**図 1** で説明する．状態 S1 からノード C の挿入とノード B の削除がほぼ同時に発生し，図で示したシーケンスでメッセージが交換されたとする．S4 で B の削除が完了するが，このときまだ D は C からの SetL メッセージを受信していないため左リンクが B を指したままとなり，左リンクを使ってメッセージを転送できない．(なお，図では SetL メッセージの処理の完了を SetLAck メッセージで通知しているが，この方法は役に立たないことがわかる．)

3.3　参照カウンタと左リンク解放通知メッセージの導入

他のノードから左リンクで参照されている間はノードを削除できないようにするため，他のノードからの左リンクによる被参照数を数える参照カウンタ (ref) と，左リンク解放通知 (UnrefL) メッセージを導入する．

挿入済みのノードの ref は定常状態では 1 である（右ノードの左リンクでのみ参照されているため）．ref は SetR メッセージを受信した場合に 1 加算し，UnrefL メッセージを受信したときに 1 減算する．SetR メッセージ受信で ref を 1 加算する理由は，(1) ノード挿入のために SetR メッセージを受信する場合，挿入ノードからの左リンク参照が増え，また (2) ノード削除のために SetR メッセージを受信する場合，削除ノードの右ノードからの左リンク参照が（いずれ）増えるためである．

© 2010 Information Processing Society of Japan

Vol.2010-DPS-144 No.1
2010/9/17

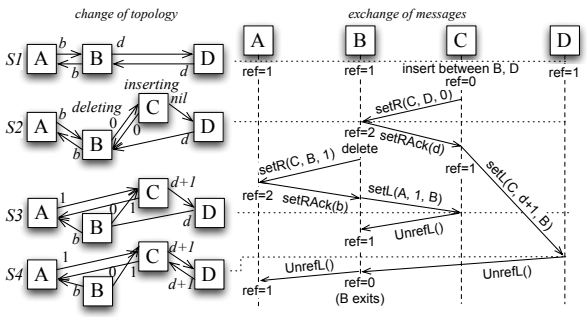

図 2 参照カウンタと左リンク解放通知を用いた例

ノードが SetL メッセージを受信したときは，変更前の左ノードに UnrefL メッセージを送信する．ただし，受信した SetL メッセージの $\mathrm{l_{newnum}}$ が x のときに，$\mathrm{l_{newnum}}$ が $x-1$ の SetL メッセージを受信しているとは限らないため,「変更前の左ノード」がわからない可能性がある．このため，変更前の左ノードは SetL メッセージの引数 $\mathrm{l_{prev}}$ で通知する．

他のノードから u への左リンクが存在する限り，u へメッセージが転送される可能性があるため，u の削除は ref が 0 になるのを待つ必要がある．ref が 0 になった場合，$u.l$ に UnrefL メッセージを送信し，$u.l$ に対する左リンクが削除されたことを通知してから u は動作を停止する．

図 2 は改良した方式を用いて図 1 の例を実行した場合の例である．状態 S1 での A–B 間，B–D 間の左右のリンク番号はそれぞれ b，d としている（リンク番号は矢印の根本に示した）．図 1 の例と異なり，D の左ノードは常に生きている点に注意せよ．

3.4 形式的記述

故障を考慮しない DDLL アルゴリズムの形式的な記述を**図 3** に示す．アルゴリズムは文献 4)，6) にならい，文献 7) の表記法と類似した方法で記述した．この表記法では，ノードの動作は変数宣言のための **var** セクション, 変数初期化のための **init** セクションと，アクションの集合で定義される．アクションは ⟨ ガード式 ⟩ → ⟨ 文 ⟩ という形をしている．複数のアクションは [] で区切られる．アクションの文はガード式が真である場合のみ実行される．アクションの実行はアトミックであり，あるアクションを実行中に同一ノード上で他のアクションが実行されることはない．あるノードで複数のアクションが実行可能な場合は，いずれかがランダムに選択されて実行される．文は上から下に実行される．代入文 (:=) は複数の変数に同時に代入できる．**skip** は現在のアクションの実行を終了する．**timeout**⟨ 式 ⟩ は，式が真であるまま一定のタイムアウト時間が経過したら真になるガード式である (具体的なタイムアウト時間はここでは指定しない)．アンダースコアから始まる変数はアクションを実行している間のみ存在する局所変数である．

変数 s はノードの状態を示し，以下の値をとる．

out リストから外れている
ins ノードを挿入するために左ノードに SetR メッセージを送信中
inswait 挿入中に SetRNak を受信し，リトライ待ち
in （少なくとも）右方向連結リストに挿入済み
del ノードを削除するために左ノードに SetR メッセージを送信中
delwait 削除中に SetRNak を受信し，リトライ待ち
grace ノード削除時に，左リンクによる被参照数が 0 になるのを待機中

なお，ノードは，状態が out になればアルゴリズムの実行を停止するものとする．

ノードが挿入，削除する際は，それぞれ A_1，A_7 から実行する．contact() は，ノード u の挿入場所を求める手続きで，返り値 (a, b) は，u の挿入場所はノード a と b の間ということを表す ($\mathrm{ord}(a, u, b) = \mathrm{true}$)．contact() の詳細は 3.5 節で述べる．なお，各アクションの実行はアトミックに行うと述べたが，contact() の実行中は他のアクションを実行してもよい (contact() は他のノードと通信する必要があり，実行時間が長い可能性がある)．contact() の実行が終了した時点で再度ガード式を評価し，真ならばアクションの実行を継続する．

初期ノード（最初に連結リストに挿入するノード）u では，$u.l = u.r = u, u.s = in, u.\mathrm{l_{num}} = u.\mathrm{r_{num}} = 0, \mathrm{ref} = 1$ としておく．

3.5 検索処理

ここでは，連結リスト上でノードの挿入位置を検索する処理を述べる（図 3 の contact() に対応）．通常の検索処理（キーを与えて対応するノードに到達する）もほぼ同様であるため省略する．検索の際は，ノードの右リンクは常に最も近い右ノードを指しているが，左リンクは常に最も近い左ノードを指しているとは限らないことに注意する．

挿入しようとするノード u は何らかの方法で $q.s = \mathrm{in}$ であるような任意のノード q を知っているものとする．ノード u の挿入位置を検索するアルゴリズムを以下に示す．なお，n は注目ノード，p は直前の注目ノードを表す．

(1) $n := q$ とする．また $q < u$ ならば $p := q.l$，そうでなければ $p := q.r$ とする．
(2) $\mathrm{ord}(n, u, n.r) = \mathrm{true} \wedge n.s \neq \mathrm{grace}$ ならば，n と $n.r$ をそれぞれ u の左ノード，右

© 2010 Information Processing Society of Japan

Vol.2010-DPS-144 No.1
2010/9/17

```
1  process u
2  var s: {out, ins, inswait, in, del, delwait, grace}
3    l, r: {pointer to a node or nil}
4    l_num, r_num: {integer or nil}
5    ref: integer
6  init s = out; l = r = nil; l_num = r_num = nil; ref = 0
7  begin
8  {Insert}
9     (A_1)  s = out ∨ timeout s = inswait →
10      (l, r) := contact(); s, l_num := ins, 0; send SetR(u, r, l_num) to l
11 [] (A_2) receive SetR(r_new, r_cur, r_newnum) from q →
12      if ((s ≠ in ∧ s ≠ delwait) ∨ r ≠ r_cur) then send SetRNak() to q
13      else send SetRAck(r_num) to q; r, r_num, ref := r_new, r_newnum, ref + 1 fi
14 [] (A_3) receive SetRAck(r_prevnum) from q →
15      if (s = ins) then s, r_num, ref := in, r_prevnum + 1, 1; send SetL(u, r_num, 1) to r
16      elseif (s = del) then s = grace; send SetL(l, r_num + 1, u) to r fi
17 [] (A_4) receive SetRNak() from q →
18      if (s = ins) then s := inswait elseif (s = del) then s := delwait fi
19 [] (A_5) receive SetL(l_new, l_newnum, l_prev) from q →
20      if (l_num < l_newnum) then l, l_num := l_new, l_newnum fi
21      send UnrefL() to l_prev
22 [] (A_6) receive UnrefL() from q →
23      ref := ref − 1
24      if (ref = 0) then send UnrefL() to l; s := out fi
25 {Delete}
26 [] (A_7) s = in ∨ timeout s = delwait →
27      if (r = u) then s := out else s := del; send SetR(r, u, r_num + 1) to l fi
28 end
```

図 3 故障を考慮しない DDLL アルゴリズム

ノードの候補とし，探索を終了する．

(3) $\mathrm{ord}(n, u, p) = \mathrm{true} \wedge n.s \neq \mathrm{grace}$ ならば，$p := n; n := n.r$ とし，ステップ 2 に戻る．

(4) $p := n; n := n.l$ とし，ステップ 2 に戻る．

状態が grace であるノードは実質的には削除されているので検索結果は返さない（ステップ 2）．また，状態が grace であるノードの右リンクは信頼できないため（右ノードは削除されて存在しない可能性がある），このようなノードの右リンクは使用しない（ステップ 3）．

左リンクをたどって検索すると，いずれ u を飛び越すはずであり（前提から u と値が等しいノードはリスト中に存在しない），飛び越した場合はそれ以降ステップ 3 の条件が満たされるため，右リンクだけを辿ることになる．右リンクを辿っていくといずれステップ 2 の条件が満たされるため，このアルゴリズムにより u の挿入位置を見つけることができる（なお，右リンクを辿るときに状態が grace であるノードを通ることはない）．

なお，このアルゴリズムで右リンクを辿るときに，状態が ins であるノードを通る可能性がある．これは，挿入時の SetRAck メッセージ到着よりも前に，既に書き変わっている左ノードの右リンクによって検索メッセージが転送される場合があるためである．また，このアルゴリズムでは u の左ノードとして状態が del であるノードを返す可能性があるが，これは意図的である（状態が del であるノードは SetRNak を受信することで状態が in に戻る可能性があるため）．

3.6 考　　察

形式的にはノード n は，以下の関数 $\mathrm{inserted}(n) = \mathrm{true}$ のときに**挿入されている**という．なお，$m^{-}(msg, n)$ は，ノード n に送信中のメッセージ msg の数である．

$$\mathrm{inserted}(n) = (n.s = in) \vee (n.s = delwait) \vee (n.s = ins \wedge m^{-}(\mathrm{SetRAck}, n) = 1) \vee (n.s = del \wedge m^{-}(\mathrm{SetRAck}, n) = 0)$$

このアルゴリズムは次の性質を有する（証明は紙数の都合で省略する）．(1) 挿入されているノードの右リンクは，常に，右方向で最も近い，挿入されているノードを指す．(2) 挿入されているノードの左リンクは，いずれ，左方向で最も近い，挿入されているノードを指す．(3) 挿入されているノードの左リンクは常に状態が out ではないノードを指す．

4. 故障を考慮したアルゴリズム

この章ではノードが故障した場合に，故障したノードをバイパスしてリンクを修復できるアルゴリズムについて述べる．なお，ビザンチン障害は想定しない．

3 章のアルゴリズムでは，SetL メッセージを受信するノードは左リンク番号が単調に増加するようにしていた．リンク修復の際にも左リンク番号を単調に増加させるためには，リンク修復は故障したノードの右ノードが主体となって行う必要がある（故障したノードの左ノードは，故障したノードの右ノードの左リンク番号を知らないため）．

ノード u の左ノードが故障した場合，次の流れでリンクを修復する．(1)u の左側で最も近い，故障していないノード v と，v の右ノード $v.r$ を得る．(2)u は左リンクを v に変更してから v に $\mathrm{SetR}(\mathrm{r_{cur}}{=}v.r, \mathrm{r_{new}}{=}u)$ メッセージを送信する（SetR のその他の引数については後で議論する）．(3)SetR メッセージを受信した v は，メッセージを前章で述べた方法で処理する．(4)u が SetRAck を受信すれば修復は完了する．SetRNak メッセージを受信した場合はしばらく待ってステップ 1 からやりなおす．

しかし，修復処理とノードの挿入などが並行して行われたり，あるいは，ネットワーク障害などの原因で，故障していないノードを誤って故障していると判定されたりする場合があ

© 2010 Information Processing Society of Japan

るため，修復処理はもう少し複雑である．

以下，詳細を述べる．なお，アルゴリズムの形式的記述は 4.7 節で示すが，本文中でも対応するアクションを (A_1) のように示す．

4.1 左側で最も近い生存ノードの発見

ノード u は左ノードの故障を検出し，また検出した場合は代わりに接続するノードを見つける必要がある．

このために，本稿では (v，$v.r$，$v.\mathrm{r_{num}}$) を返す手続き getLiveLeftStat() が利用できることを想定する．ここで v は，u の左側で（見つけられる範囲で）最も近い，故障していない，挿入されているノードである．

v を見つけるには次のようにすればよい．u の左側の，状態が in あるいは delwait であるノードを起点とし，そこから右リンクを辿る．故障しているノードに到達するか，u に到達するか，u を飛び越した場合，その 1 つ前のノードを v とする．$v.r$，$v.\mathrm{r_{num}}$ は v と通信することで得られる．起点となるノードを求めるには，例えば，(1) 連結リストを使用している上位のデータ構造（オーバーレイネットワークなど）を利用する，(2) 各ノードが十分な数の左側ノードへのポインタ（近隣ノード集合と呼ぶ）を維持し，これを利用する，などの方法が考えられるが，紙数の関係で詳細は割愛する．

4.2 リンク修復処理

ノード u のリンク修復処理について述べる．なお，u が挿入中 (ins) あるいは削除中 (del) の場合は，リンク修復処理は行わない．この場合については 4.6 節で扱う．また，$u.s$ が inswait の場合も u はまだリストに挿入されていないため修復処理は行わない．

その他の場合は定期的に以下の処理を行う (A_{11})．

(1) $(v, v.r, v.\mathrm{r_{num}})$:=getLiveLeftStat() を実行する．

(2) u が grace 期間中に左ノードが故障した場合（つまり，$u.s = grace \wedge v \neq l$ の場合），$u.l$ を v に変更する．これにより，左方向連結リストの接続性を確保する．なお，v は新たに u の左リンクで参照されることになるが，v の参照カウンタは増加させていないため，この処理を実行したあとで u が削除完了したときは最後の UnrefL メッセージは送信しない．

(3) $(v = u.l \wedge v.r = u \wedge v.\mathrm{r_{num}} = l.\mathrm{l_{num}})$ の場合，v–u 間のリンクおよびリンク番号の関係は正常であるため，何もしない．

(4) その他の場合はリンクを修復する．u は $u.l$ を v に変更し，v に SetR($\mathrm{r_{cur}} = v.r, \mathrm{r_{new}} = u$) メッセージを送信する．なお，このときの，$\mathrm{r_{newnum}}$ の値については，次の 4.3 節で議論する．この SetR メッセージに対する応答 (SetRAck/SetRNak) が到着するか，タイムアウトするまでは新たな修復処理は実行しない (A_3, A_4, A_{12})．

v が SetR メッセージを受信した時点で，v の右に別のノードが挿入されている可能性があるが，この場合 v は SetRNak を返すので u は修復処理をリトライする．

4.3 リンク番号の拡張

上のステップ 4 で，u は v に SetR メッセージを送信して v–u 間を接続するが，このリンクに対する左右のリンク番号（u が v に送信する SetR メッセージの引数 $\mathrm{r_{newnum}}$ の値でもある）について考える．

ノード A, B, C がこの順に並んでいるとする．また，B–C 間のリンクに対する左右のリンク番号を c とする ($B.\mathrm{r_{num}} = C.\mathrm{l_{num}} = c$)．ここで B が故障した場合を考える．C は getLiveLeftStat() により A を発見する．ここで A–C 間のリンクに対する左右のリンク番号を単純に $c+1$ にすると次の問題がある．

B が故障する直前に，B と C の間にノード X が挿入されたとする．(C は X からの SetL メッセージを受信する前に getLiveLeftStat() を実行し，X は発見できなかったとする)．X は B から SetRAck によりリンク番号 c を受け取るため，X の右リンク番号も $c+1$ となる．つまり，同一の右ノード (C)，右リンク番号 $(c+1)$ を持つ複数のノード (A, X) が存在することになるため，以後 C が受け取る SetL メッセージに正しくシーケンス番号 ($\mathrm{l_{newnum}}$) を付与できない．

この問題を避けるため，リンク番号を (g, s) という形式に拡張する．s はいままでのリンク番号に対応する．g はリンクを修復した回数を表し，初期状態では 0 である．以下，ノードの左右のリンク番号は (g, s) 形式とする．リンク番号 x と y に対し，以下の規則により等値関係，大小関係を定義する．

$$x = y \text{ iff } (x.g = y.g) \wedge (x.s = y.s)$$
$$x < y \text{ iff } (x.g < y.g) \vee (x.g = y.g \wedge x.s < y.s)$$

また，リンク番号 x に対し $x.\mathrm{gnext}() = (x.g + 1, 0)$, $x.\mathrm{next}() = (x.g, x.s + 1)$ と定義する．

u が左リンクを修復して v と接続する場合，修復前の $u.\mathrm{l_{num}}$ を w とすると，u–v 間の左右のリンク番号は，$w.gnext()$ とする（つまり，u は $u.\mathrm{l_{num}} := u.\mathrm{l_{num}}.gnext()$ としてから v に SetR メッセージを送信する）．

上の例で B–C 間のリンク番号を (c_g, c_s) とすると，X の右リンク番号は $(c_g, c_s + 1)$ となるが，リンク修復後の C の左リンク番号は $(c_g + 1, 0)$ となる．リンク修復後に X からの SetL メッセージを C が受信しても，リンク番号の大小関係からメッセージは無視され

© 2010 Information Processing Society of Japan

る[*1]．このため，リンク修復前の状態には戻らない．

4.4 修復時の参照カウンタの扱い

次に，修復時の参照カウンタ (ref) の扱いについて考える．

ノード A, B, C がこの順に並んでいるとする．B が故障した場合，C が A に送信するリンク修復のための SetR メッセージでは A の ref は増加させない．これは A から見ると B の左リンクが消失し，代わりに C の左リンクによって参照されるためである．SetR メッセージによって ref を加算する場合と加算しない場合があるため，ref に加算する値は SetR メッセージの引数 incr で決定する (A_2)．

また，前章のアルゴリズムではノードを削除する際，grace 状態となってから ref が 0 になるのを待機していたが，UnrefL メッセージを送信すべきノードが故障した場合，いつまでたっても ref が 0 にならない場合がある．このため，grace 状態になってから一定時間経過した後は強制的に削除完了とする (A_{10})．

4.5 ノード故障誤検出からの修復

ネットワーク障害などの理由により，実際は故障していないノード x が故障していると判断される可能性がある．この場合，x は連結リストから外れてしまう．例えば**図 4** の状態 S1 でノード E がノード C を故障していると誤って判断すると状態 S2 になる．上述の方法で，このような状態からも修復できることを示す．(なお，getLiveLeftStat() が，最も近い挿入されているノード x を見逃した場合も同様に x は連結リストから外れてしまうが，この場合も同様に修復できる．)

図の状態 S3 は，状態 S2 からノード A と E の間にノード B が挿入され，さらにノード C と E の間にノード D が挿入された状態である（D から E に送信された SetL メッセージはリンク修復回数 (g) の違いにより無視されている）．その後，C が getLiveLeftStat() を実行したとすると，B が見つかるため，C の左リンクを B に変更し (S4)，SetR メッセージを B に送信する (S5)．この状態では E の左リンクが正しいノードを指していないが，E が getLiveLeftStat() を実行すると D が見つかるため，D に SetR メッセージを送信して修復が完了する (S6)．

4.6 挿入・削除時のノード故障

ノード u を挿入する際に，u の左ノードとなるノード a が故障し，SetR メッセージへの応答 (SetRAck/SetRNak) がタイムアウトした場合，n は再度挿入位置を検索し，挿入処理をリトライする (A_7)．いずれ a の右ノードが a をバイパスするようにリンクを修復するため，u を挿入できるようになる．ただし，リトライする前に，u の左リンク番号の修復回数 (g) を増加させる必要がある．これは，最初の SetR メッセージは a に届いているが応答を受信していないという場合があるためである．

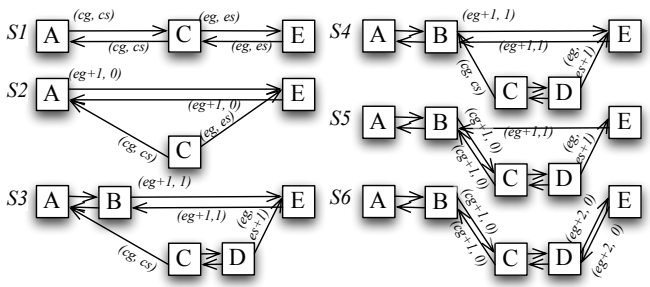

図 4 ノード故障誤検出からの修復例

ノード u を削除する際に，左ノード $u.l$ が故障し，SetR メッセージへの応答がタイムアウトした場合，SetRAck を受信した場合と同様に扱う．すなわち，SetL メッセージを右ノード $u.r$ に送信し，右ノードの左リンクを $u.l$ に更新する (A_9)．リンクの修復は右ノードに任せる．これは，u は勝手に右ノードの左リンク番号の修復回数 (g) を増加させることができないためである（u の右ノードは独立して g を増加させている可能性がある）．

4.7 形式的記述

故障を考慮する DDLL アルゴリズムの形式的な記述を**図 5** に示す．getLiveLeftStat() については 4.1 節で述べた．contact() 同様，getLiveLeftStat() 実行中も他のアクションを実行してもよい．

なお，ノード u が SetR に対する応答メッセージ（SetRAck と SetRNak）を受信した際，それが u が送った最新の SetR メッセージに対する応答でなければ破棄する必要がある．このためには SetR メッセージにシーケンス番号を含め，SetR メッセージを受信したノードが応答メッセージに同じ番号を含めて返すことでチェックできる（図 5 では省略している）．

4.8 ノードの再挿入について

ノード u が，一度挿入した後，削除あるいは故障し，再度挿入しようとする場合，u が完全に挿入される前から左右のリンクや近隣ノード集合によって u を参照しているノー

*1 このとき，X は正常にリストに挿入されていない．リンク修復回数 (g) の違いにより SetL メッセージが無視されたことを X に通知することで，X は速やかにリンク修復処理を実行して正常に挿入できるようになる．

© 2010 Information Processing Society of Japan

ドが存在している場合がある．再挿入前の u の左右のリンクは正しいとは限らないため，getLiveLeftStat() やノードの検索に u を用いると正しく動作しない可能性がある．この問題に対処するための単純な方法は，再挿入時にキーを変更することである．それにはキーの乱数部分（2 章参照）を変更すればよい．

5. おわりに

本稿では，分散双方向連結リストを構築・維持するための自律分散アルゴリズム DDLL について述べた．DDLL では，複数のノードが並行して挿入・削除する場合でも，連結リストの構造は常に維持される．また，分散排他制御を使用しないため，ノード故障から容易に修復できる．これらの性質により，DDLL は特に構造化オーバーレイネットワークの実装に有用である．また，DDLL で用いた，(1) 右リンク更新処理を条件付きで行う，(2) 右リンクを更新する際に左リンク更新メッセージのシーケンシャル番号を決定する，(3) シーケンシャル番号にリンク修復回数を格納する，といった技法は，他の分散データ構造の実装でも有効と思われる．

今後の課題としては，DDLL を用いて各種の構造化オーバーレイアルゴリズムを実装し，特に churn 発生時の性能を解析することが挙げられる．

参考文献

1) Stoica, I. et al.: Chord: A Scalable Peer-to-Peer Lookup Protocol for Internet Applications, *IEEE/ACM Trans. on Networking*, Vol.11, No.1, pp.17–32 (2003).
2) Schütt, T. et al.: Scalaris: reliable transactional p2p key/value store, *ERLANG '08: Proc. of 7th ACM SIGPLAN workshop on ERLANG*, ACM, pp.41–48 (2008).
3) Aspnes, J. and Shah, G.: Skip graphs, *ACM Trans. on Algorithms*, Vol.3, No.4, pp.1–25 (2007).
4) Shaker, A. and Reeves, D.: Self-Stabilizing Structured Ring Topology P2P Systems, *Proc. of 5th IEEE Intl. Conf. on P2P Computing*, IEEE Computer Society, pp.39–46 (2005).
5) Lynch, N. et al.: Atomic Data Access in Content Addressable Networks: A Position Paper, *Proc. of 1st Intl. Workshop on P2P Systems*, pp.295–305 (2002).
6) Li, X. et al.: Active and Concurrent Topology Maintenance, *Proc. of 18th Ann. Conf. on Distributed Computing (DISC2004)*, Springer, pp.320–334 (2004).
7) Mohamed G. Gouda: *Elements of Network Protocol Design*, John Wiley and Sons (1998).

```
1  process u
2  var s: {out, ins, inswait, in, del, delwait, grace}
3    l, r: {pointer to a node or nil}
4    l_num, r_num: (g:integer, s:integer)
5    ref: integer; fixing, lastUnrefL: boolean
6  init s = out; l = r = nil; l_num = r_num = (0, 0); ref = 0;
7    fixing = false; lastUnrefL = true
8  begin
9  {Insert}
10    (A_1) s = out ∨ timeout s = inswait →
11      (l, r) := contact(); s, l_num.s := ins, 0; send SetR(u, r, l_num, 1) to l
12 [] (A_2) receive SetR(r_new, r_cur, r_newnum, incr) from q →
13      if ((s ≠ in ∧ s ≠ delwait) ∨ r ≠ r_cur) then send SetRNak() to q
14      else send SetRAck(r_num) to q; r, r_num, ref := r_new, r_newnum, ref + incr fi
15 [] (A_3) receive SetRAck(r_prevnum) from q →
16      if (s = ins) then s, r_num, ref := in, r_prevnum.next(), 1; send SetL(u, r_num, 1) to r
17      elseif (s = del) then s := grace; send SetL(l, r_num.next(), u) to r
18      elseif (fixing) then fixing := false fi
19 [] (A_4) receive SetRNak() from q →
20      if (s = ins) then s := inswait elseif (s = del) then s := delwait
21      elseif (fixing) then fixing := false fi
22 [] (A_5) receive SetL(l_new, l_newnum, l_prev) from q →
23      if (l_num < l_newnum) then l, l_num := l_new, l_newnum fi
24      send UnrefL() to l_prev
25 [] (A_6) receive UnrefL() from q →
26      ref := ref − 1
27      if (ref = 0) then
28        if (lastUnrefL) send UnrefL() to l fi
29        s := out fi
30 [] (A_7) timeout s = ins → {left node might fail while inserting}
31      (l, r) := contact(); l_num := l_num.gnext(); send SetR(u, r, l_num, 1) to l
32 {Delete}
33 [] (A_8) (s = in ∨ timeout s = delwait) ∧ ¬fixing) →
34      if (r = u) then s := out else s := del; send SetR(r, u, r_num.next(), 1) to l fi
35 [] (A_9) timeout s = del → {left node might fail while deleting}
36      if (r = u) then s := out
37      else s, lastUnrefL := grace, false; send SetL(l, r_num.next(), u) to r fi
38 [] (A_10) timeout s = grace → send UnrefL() to l; s := out
39 {Fix}
40 [] (A_11) (s = in ∨ s = delwait ∨ s = grace) ∧ ¬fixing → {execute periodically}
41      (v, vr, v_rnum) := getLiveLeftStat()
42      if (s = grace) then
43        if (l ≠ v) then l, lastUnrefL := v, false fi
44        skip
45      fi
46      if (v = l ∧ vr = u ∧ v_rnum = l_num) then skip fi {left link is consistent}
47      l, l_num, fixing := v, l_num.gnext(), true; send SetR(u, vr, l_num, 0) to l
48 [] (A_12) timeout fixing →
49     fixing := false
```

図 5 故障を考慮した DDLL アルゴリズム

© 2010 Information Processing Society of Japan