並列オブジェクト指向言語 LGO の故障回復機構

野里貴仁[†] 杉本 明[†] 阿部 茂[††]

分散システムの開発では故障回復のプログラミングが必要であり，ソフトウェアの複雑化の一因となっていた．本論文では，分散 Linda システムにおける，プログラマによる故障回復のための記述を必要としない，プロセスの故障回復方法について述べている．われわれの方法は，故障によって，動かなくなったプロセスをシステムが自動的に再実行し，プロセスの状態を故障前と全く同一にすることで，故障から回復する方式である．その再実行方式の特徴は，履歴を用いて行うことにある．正しく故障前の状態になるように，再実行の時には，故障前に残しておいた履歴を使用する．この履歴は分散タプルスペース上に残している．再実行をしているプロセスは履歴だけに依存しているので，他のプロセスの状態を知ることも，他のプロセスに影響を与えることもない．したがって，故障からの回復の際には，他のプロセスの状態と矛盾した状態になることはなく，他のプロセスが実行をやり直すこともない．また，タプルスペースの一貫性を保つための通信と同時に，履歴を残すので，通信量が大きく増加することはない．論文中では Linda をもとにした並列オブジェクト指向言語 LGO を用いて，故障回復を説明する．そして，UNIX ネットワーク上に LGO の実行システムを実現し，そこでの実験から，履歴を残すことによって，通信コストが大きく増えることがないことを示している．

A Recovery Method of Object Oriented Parallel Language LGO

Takahito Nozato,[†] Akira Sugimoto[†] and Shigeru Abe[††]

This paper describes a recovery method of a distributed Linda system. In a distributed system, which is more complex than a single computer system, it is required to eliminate a burden of writing code for recovery. Our recovery method is to restart dead processes which run on a failure node. This replay is done automatically. The novel feature of our recovery method is that the replay is based on history crseated on a distributed Tuple Space. The replayed process repeats the same execution as executed before the failure. To do that, the replay is based on history created at normal run-time. Because the replayed process is depend on only the history, the replayed process require no information of other process and affects other processes no more. Therefore, during the recovery time, there are no needs to rollback processes. And besides, because the distributed Tuple Space stores the history as a side effect of tuple management, little communication for recovering is required additionally. We explain the recovery method by using Linda based parallel language LGO. And from experiences on UNIX Workstations, we show that creating history does not increase the communication cost.

1. はじめに

近年ネットワークの発達により，複数計算機のネットワークからなる分散システムを作ることが容易になった．このために多くの分野で分散システムが増加している．産業システムも例外ではなく，一極集中型の構成から，分散化された環境に移行している．このような分散システムを効率良く構築するために，分散処理のためのモデルや言語が多く提案されている[2),13)]．それらの中で最近注目を集めているものに Linda[1),6)] がある．

Linda は当初，並列プログラムの記述を容易にするために設計された[1)]が，現在では，並列計算機上だけでなく，計算機ネットワークなどの分散環境上でも多くの処理系が実現されている[8),14)]．Linda の特徴はそのプロセス間通信にある．送り手のプロセスは，プロセスの共有空間（タプルスペースと呼ばれる）に生成したデータ（タプルと呼ばれる）を加えるだけである．そのデータを，誰が，いつ受け取るかなどを一切指定しない．受け手のプロセスは，タプルをタプルスペースから取り除く，または，読み込むことで送り手からの情報を得る．このときも，送り手が誰であるかを指定しない．このようにプロセス同士のつながりは重要でないので，プログラムは複数のプロセスの実行を把握する必要がなく，一つのプロセスの実行に注目して，プログラムを書くだけで良い．このために，プ

† 三菱電機株式会社中央研究所
Central Research Laboratory, Mitsubishi Electric Corporation
†† 三菱電機株式会社産業システム研究所
Industrial Electronic & Systems Laboratory, Mitsubishi Electric Corporation

ログラマの負担が小さくなり，分散プログラムの開発が容易になると考えられる．

しかしながら，大規模な産業システムを Linda を使って記述するには，次のことが問題となる．

（1） 耐故障性がないこと．

産業システムでは高い信頼性が要求されるが，Linda モデル自身は故障に対して何も対策を施していない．このため，産業システムの構築には，故障の対策を記述することが必要になり，プログラムが複雑になってしまう．

（2） 一つのタプルスペースをすべてのプロセスが共有していること．

システムにはタプルスペースが一つしかないので，関連のないタプルが同時にタプルスペース上に存在することになる．このため，タプルの受け取りに間違いが起こり，システム全体に影響を与えてしまう．特に，大規模なシステムでは，多くのプロセスが存在するので，このような間違いが発生しやすく，そして，発見が困難となる．

われわれはこれらの問題を解決するために，LGO (Large Graind Object) を設計した．

LGO は Linda に大規模産業システムの構築に必要な機能を付加した言語である．LGO では，故障が発生しても実行を継続することを保証しているので，プログラマは故障対策を記述する必要がない．また，タプルスペースを分割することで，誤ったタプルの授受を防いでいる．

本論文では LGO の故障回復について述べる．LGO は複数の Linda モデルから構成されているので，われわれの故障回復方式は通常の Linda にも適用することができる．われわれの故障回復方式の特徴は，再実行に必要な情報を分散タプルスペースの一貫性管理と共に残すことより，実行の遅れや通信量の増加を抑えていること，および，Linda のプロセスの実行を解析すること，プロセスの状態（変数の値やプログラムカウンタなど実行に必要な局所的な情報）を再実行のために残す必要をなくしたことである．

故障が起きても，実行を続けるシステムについては従来から多くの研究が行われてきた[3),5),10),11)]．文献11)はフォールトトレラントなタプルスペースの実現について述べている．しかし，これはプロセスに故障が起きないものと仮定している．文献3)，5)，10)は Linda システムではなく，メッセージを使った通信をするプロセスの故障回復方式について述べたものである．文献10)はロールバック（すでに行った計算をキャンセルし，以前の状態からやり直すこと）が発生する回復方式であり，文献3)，5)はロールバックが発生しない方式である．産業システムではロールバックが発生する故障回復方式を用いることはできない．なぜならば，産業システムではすでに実行してしまった制御をやり直すことができないからである．一方，ロールバックが発生しない文献3)，5)の方式はどちらも，再実行時に必要な実行の記録を別ノード上に，システム本来の計算と同期して残している．文献11)のフォールトトレラントタプルスペースではタプルスペースのレプリカを用いており，タプルの操作の度に一貫性を保つための遅れが発生する．このために文献3),5)の方式とフォールトトレラントタプルスペースを用いると，本来の計算が故障回復のために大幅に遅れてしまう．

LGO ではフォールトトレラントタプルスペースとプロセスの再実行を用いて，耐故障性を実現している．プロセスの故障回復では，ロールバックを起こさないように，通常時に残しておいた実行記録を用いて再実行する．しかし，われわれの方式ではフォールトトレラントタプルスペースの一貫性管理と同時に，実行記録を残すことで，通信量の増加と計算の遅れを防いでいる．また，文献3)，5)では再実行に要する時間を短縮するためにプロセスの状態を残す必要があるが，われわれの方式は，Linda のプロセスの特徴を利用することで，プロセスの状態を残す必要をなくしている．

本論文では，まず最初に LGO について述べた後，故障回復の機構について詳述する．以下2章で LGO について説明し，3章で LGO の実現機構の概要を述べる．4章では回復機構について詳細を述べ，5章で通信量についての評価を行う．

2. LGO

LGO は，Linda を大規模産業システムの記述に用いるために設計された言語である．以下の節では Linda を説明し，Linda で用いられるマスタワーカパラダイムについて述べる．それから，マルタワーカパラダイムとの関連を中心に LGO について説明し，LGO の耐故障性について述べる．

2.1 Linda

Linda モデルは並列計算のモデルの一種であり，並列に動くプロセスが通信をしながら，計算を進める．

この通信はタプルスペースと呼ばれる仮想的な共有メモリを使って，タプルと呼ばれるデータをやりとりすることで行う．タプルは型のついたフィールドの並びである．フィールドには文字列などのデータや変数が入る．タプルの例として("X", 1, "Tom") や ("begin") などがある．

Linda モデルでのプロセス間の通信は，送り手がタプルスペースにタプルを加え，受け手がタプルを読み込む（または取り除く）ことで行う．送り手はタプルを誰が読み込むかを指定する必要はない．同様に，受け手はタプルを誰が加えたかを指定する必要がない．このように，Linda モデルでの通信は，明示的な通信先を必要としない．

タプルに対する操作には次の3つが用意されている*.

out(t) タプルスペースにタプル t を加える．

in(s) s とマッチするタプルがあれば，それをタプルスペースから取り除く．このとき s に仮引数 (Linda では変数の前に？を付けて示す) があればそれに対応する値が代入される．マッチするタプルがない場合にはそのようなタプルが現れるまで実行を中断する．

read(s) タプルスペースからタプルを取り除かないことを除いて in と同じである．

プロセスはタプルを加えること，取り除くこと，読むことしかできない．タプルを直接変更する操作がないために，タプルの変更はタプルを取り除き，新たにタプルを加えることで行う．タプルの操作に排他制御の機構が組み込まれることになるので，プログラマが明示的にロックをかける必要がない．また，タプルの指定はパターンマッチングを使って行われ，パターンに対応するタプルがない場合には，対応するタプルが現れるまでプロセスの実行がブロックされる．このため，同期を自然な形で記述できる．

2.2 マスタワーカパラダイム

マスタワーカパラダイムとは，マスタプロセスが計算の初期化を行い，同型のワーカプロセスの集まりがその計算を実行するプログラミングスタイルのことである[6]．ワーカプロセスはその計算のどの段階でもこなすことができる．本論文では計算の各段階のことをタスクと呼ぶ．ワーカプロセスがどのタスクを実行するかは，タプルスペースからタスク割り当てのタプルを取り出して決める．そして，そのタスクの実行を終えると，次のタスク割り当てのタプルを取り出し，タスクを実行する．ワーカプロセスはタスクがなくなるまで，これを繰り返す．

ワーカプロセスの特徴は，タスクの実行がタプルスペースだけに依存することである．このために，同型のワーカプロセスならば，どのワーカプロセスがタスクを実行しても構わない．このようなワーカプロセスの集まりが計算を行うマスタワーカパラダイムは，動的な負荷分散を自然な形で実現している．なぜならば，負荷が軽いノードでは，負荷の重いノードに比べてタスクの処理が早くなり，そのぶん多くのタスクを実行するようになるからである．すなわち，負荷の軽いノードがより多くのタスクを取り出すことで，全体の負荷のバランスをとっている．

Linda のプログラミングスタイルは，マスターワーカパラダイムだけでなく，それ以外にも存在する．しかし，マスタワーカパラダイム以外で書かれたプログラムを，マスタワーカパラダイムを用いて書き直すことが可能である．しかも，その場合でも実行効率の低下はあまり見られない[6]ので，マスタワーカパラダイムを Linda の中心的なプログラミングスタイルと考えて良い．

2.3 LGO

LGO は，一つの Linda システムをオブジェクト

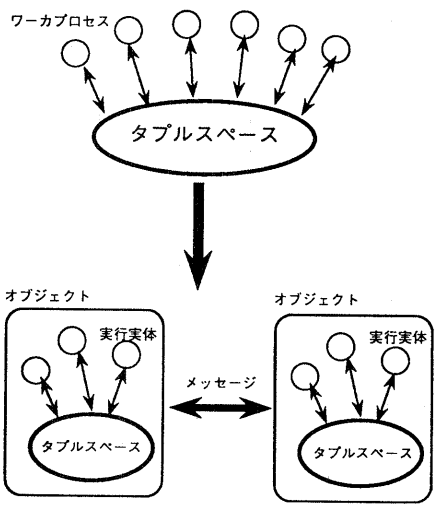


図 1 タプルスペースの分割
Fig. 1 Dividing a tuple space.

* 通常の out, in, read 以外に eval と呼ばれる操作がある．eval はタプルを別プロセスにより評価しその結果をタプルスペースに加えるものである．LGO では eval と同等なことを後述するメッセージを用いて実現しているので，eval はない．

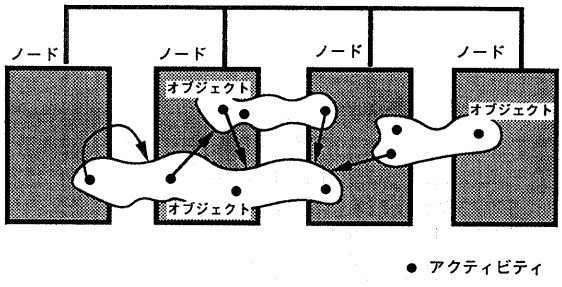


図 2 ノードにまたがるオブジェクト
Fig. 2 An object resided in inter-nodes.

とし，このようなオブジェクトが並列に動き，互いにメッセージのやりとりをしながら計算を進める並列オブジェクト指向言語である．一つの Linda システムであるオブジェクトとは，そのオブジェクトの内部状態がタプルスペースであり，そのオブジェクトの内部では，並列に動く複数の実行実体が，タプルスペースを使った情報のやりとりしている．

内部状態がタプルスペースであるオブジェクトを用いることで，通常の Linda モデルでは一つであったタプルスペースを，オブジェクトの数に分割している．この分割されたタプルスペース，すなわち，個々のオブジェクトの内部状態の中には，関連のあるタプルしか存在しない．そして，オブジェクト間のやりとりはメッセージを用いる以外に手段がないので，関連のないタプルを誤って受け取る危険がなくなる（**図1**）．

また，LGO のオブジェクトは，オブジェクト内部での並列性[7]を持っている．そして，この並列性による速度の向上を得るために，オブジェクト内の実行実体を複数のノード上で動かしている．このために，内部状態のタプルスペースは複数ノード上での分散タプルスペースとなっている．すなわち，LGO のオブジェクトは，ノードにまたがる粒度の粗いオブジェクトである（**図2**）．

オブジェクト間のやりとりに使うメッセージは，ワーカプロセスにおけるタスク割り当てに用いられるタプルに相当する．メッセージは，送り先オブジェクト名とタスク名からなる．言語にメッセージを採り入れ，タプルと区別することで，タスク割り当てのタプル（タスク用タプルと呼ぶ）と通常の計算のためのタプル（計算用タプルと呼ぶ）を間違える危険がなくなる．メッセージを受け取ったオブジェクトは，メッセージの処理をする実行実体を新たに生成する．この実行実体をアクティビティと呼ぶ．アクティビティは，ワーカプロセスでのタスクの実行に対応する．動的にアクティビティを生成しているので，オブジェクトに到着したメッセージは待たされることなく処理される．このことは4.3節で述べるアクティビティの再実行の際に有用である．

メッセージを扱うために，LGO では send と call の操作を用意している．send はメッセージを送った後，すぐに次の計算を実行する．call はメッセージを送った後，返答を受け取るまで次の計算を実行しない．呼ばれた側では return (value) によって value を送り手に返答を返す．

2.4 LGO の耐故障性

LGO は耐故障性を持つ言語なので，プログラマは故障についての記述を行う必要が全くない．LGO の耐故障性はタプルスペースとアクティビティのそれぞれの耐故障性に分けて実現している．

タプルスペースの耐故障性は，レプリカを用意することで実現している．タプルスペースのレプリカを複数のノードで動かし，レプリカ同士が常に同じ内容を保つ．そして，故障が発生すれば，故障を起こしていないノードで動いているレプリカだけを用いて計算を続ける．

アクティビティの耐故障性は，故障したときに動いていたアクティビティを別のノードで再実行することで実現している．この再実行はオブジェクトが自動的に行う．アクティビティの再実行は，その生成時から始まる．アクティビティは，ワーカプロセスでのタスクの実行に相当するので，その寿命は短く，生成時から再実行を始めても，再実行に要する時間は短い．このために，アクティビティの状態を残し，再実行に要する時間を短縮することは必要でない．

故障前と同一の実行を再現するために，LGO では，通常のタスク実行時にタプルの操作とメッセージの送受信についての記録を残し，再実行時にこの記録を用いている．この記録のことを履歴と呼ぶ．履歴の残し方と再実行時の使い方については4章で詳しく述べる．

3. 実現の概要

本章では，故障回復の詳細を述べる準備として，LGO の実現の概要について述べる．対象とするハードウェアは LAN によってつながれた計算機である．通信については，信頼性が高く，エラーの発生や二重配送がなく，通信の順序が保たれること[9]，そして，

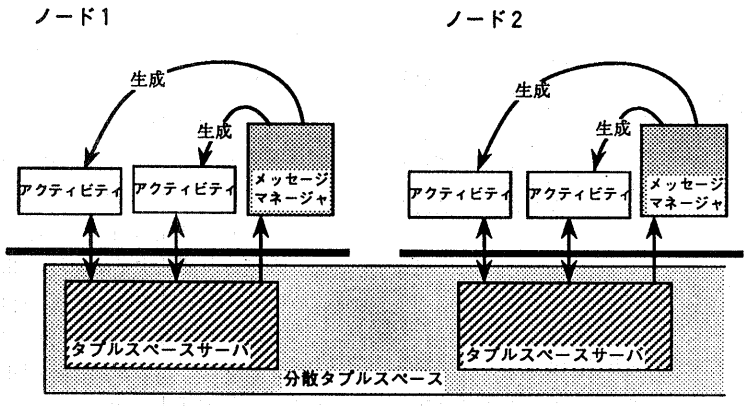


図 3 タプルスペースサーバとメッセージマネージャ
Fig. 3 Tuple space server and message manager.

アトミックな放送[4]が可能であることを仮定している．アトミックな放送とは，あるノードが送ったデータをすべてのノードが受け取るか，あるいは，すべてのノードに届かないかの，どちらかとなる放送のことである．この仮定は4章で説明するノード間での一貫性の実現で用いる．

LGO の実行システムは，互いに通信を行うプロセスの集まりである．これらのプロセスはタプルスペースサーバとメッセージマネージャの2種類に分かれる．タプルスペースサーバは分散タプルスペースを実現するプロセスであり，メッセージマネージャは，アクティビティの生成を実現するプロセスである（**図3**）．

3.1 タプルスペースサーバ

タプルスペースサーバは分散タプルスペースを実現し，オブジェクトの内部状態とメッセージをノード間で共有するためのプロセスである．タプルスペースサーバはメッセージもタプルとして管理している．すなわち，LGO では，プログラミングのレベルではメッセージを用いているが，実現レベルではメッセージをタプルとして扱っている．その理由は，LGO のメッセージは，ワーカプロセスでのタスク用タプルに対応すること，および，4章で述べるように，再実行に必要な履歴をタプルスペースサーバが管理しているが，この際に，メッセージについての履歴も一括して管理できるようにするためである．

タプルスペースサーバはすべてのノードに一つあり，各ノードのタプルスペースサーバが常にその内容を同一に保つことで，ノード間での共有を実現している．そして，各ノードのタプルスペースサーバは，同じノードで動いているメッセージマネージャとアクティビティからリクエストを受け取り，そのリクエストに従って，タプルの生成や削除を行う．タプルスペースサーバ同士はその内容を同一に保つので，あるノードでタプルを生成，削除すれば，他のすべてのノードでも，タプルを生成，削除する．ノード間の同一性のために，すべてのタプルはシステム全体で一意な ID を持っている．

タプルスペースサーバの中には，次の2種類のタプルスペースがある．

（1） オブジェクトの内部状態用のタプルスペース
（2） メッセージ用のタプルスペース

(1)は，計算用タプルを扱うタプルスペースとメッセージの操作 call の返答値を変換したタプルを扱うタプルスペースに分かれている．call の返答値もメッセージと同じように，タプルスペースサーバが一括して管理する必要があるので，一度タプルに変換した後に(1)で管理する．(2)はタスク用タプルを扱う．タプルスペースをこのように使い分けることで，タプルの誤用を防ぐことを実現している．(1)はオブジェクトごとにあり，(2)は一つだけある（**図4**）．

タプルの操作に対するリクエストには，**out**，**in**，**read** の3つがある*．

これらはタプルの操作 out，in，read にそれぞれ対応する．一方，メッセージの操作に対するリクエストには，**send**，**receive** があり，返答を扱うリクエストには **return** がある．メッセージの操作 send は **send**

* 以降ではリクリスト名は太字を使用して **out** のように表記し，操作名はローマン体を使用して out のように表記する．

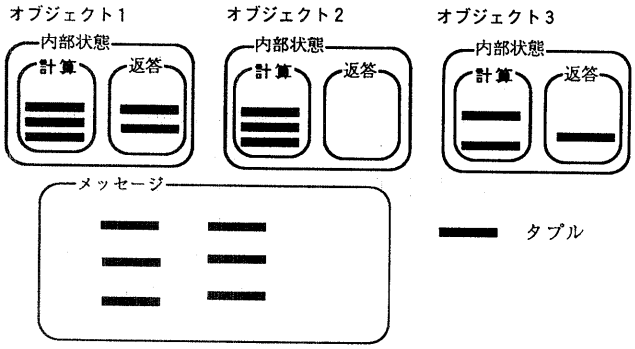


図 4 タプルスペースサーバ内のタプルスペース
Fig. 4 A tuple space on each node.

に対応し，call は **send** を送り，その直後に **receive** を送ることに対応する．そして，メッセージマネージャが送る唯一のリクエストとして **message-in** がある．**send** と **return** は **out** の変形であり，それぞれ返答用のタプルスペースとメッセージ用のタプルスペースにタプルとタスク用タプルを加える．**receive** と **message-in** は **in** の変形であり，それぞれ返答用のタプルスペースとメッセージ用のタプルスペースからタプルとタスク用タプルを取り除く．

3.2 メッセージマネージャ

メッセージマネージャはタプルスペースサーバからタスク用タプルを一つ取り出す．これはタプルスペースサーバに対してリクエスト **message-in** を送り，その結果を得ることで行っている．そして，タプルスペースサーバからの結果が，どのオブジェクトに送られたタスク用タプルかを判定し，そのオブジェクトのアクティビティを作る．タスク用タプルとアクティビティは一対一に対応するので，タスク用タプルの ID をアクティビティの識別に使う．このタスク用タプルの ID を単にアクティビティ ID と呼ぶ．アクティビティ ID は 4.4 節のノード間での一貫性の実現に用いる．

4. 回 復 方 式

本章では LGO における故障回復方式について述べる．ここで故障としてノード故障を考え，故障を起こしたノードはそのまま停止するものとする．故障後に正しく計算を続けるために，故障したノードで動いていたアクティビティを別のノードで再実行する．アクティビティの再実行を故障前と全く同じに行うため，アクティビティの通常実行時にはその履歴を残している．

LGO の故障回復方式の特徴は，回復のための機構がタプルスペースサーバ（以下本章ではサーバと略す）内に隠蔽されていることである．すなわち，アクティビティは再実行時でも通常の実行時でも同じリクテストをサーバに送る．そして，実行履歴はすべてサーバ内で管理され，ノード間で一貫性を保って共有される．しかも，履歴の共有はタプルスペースの一貫性管理における通信と同時に行われるので，通信のコストは大きくならず，計算の大幅な遅れはない．

以下次節では故障回復の概要について述べ，その次にサーバ内での履歴の実現手法を示し，最後にノード間で内部状態と履歴を共有するための操作の詳細を述べる．

4.1 故障回復の概要

故障回復はノードの故障を発見することから始まる．これはサーバが検出する．サーバ同士は互いに通信を行っており，通信できないサーバが見つかれば，そのノードは故障したと見なす．故障発見後にサーバは，故障したノードで動いていたアクティビティに対応するタスク用タプルを再び取り出すことができるようにする．各ノードのメッセージマネージャは通常のタスク用タプルと同様に，これを取り出し，アクティビティを再生成する．

アクティビティを再実行する際に問題となるのは，一度タプルスペースに加えたタプルをもう一度加えることや，再実行時に違うタプルを読み込むことなどである．同じタプルを加えることは，タプルスペースの内容を変更することとなり，再実行が他のアクティビティの実行に影響を与えてしまう．違うタプルを読み込むと，再実行の結果が故障前と違ってしまう．これはメッセージ，すなわち，タスク用タプルに対しても同様である．

このような問題に対処するため，サーバは通常のアクティビティ実行時に，アクティビティからのリクエストの記録を履歴として残している．そして，再実行のアクティビティからのリクエストに対しては，履歴を使って結果を返す．このために，アクティビティの再実行がタプルスペースに変更を加えることがなく，アクティビティの再実行の結果が故障前と異なることもない．再実行は，故障前に実行した最後のタプルの操作を行った時点で完了する．その後は通常のアクテ

ィビティとなり，サーバは履歴を追加していく．履歴はアクティビティが正常に終了すると抹消される．

4.2 タペルスペースサーバ内での履歴の実現

サーバ内ではタプル（タスク用タプルを含む）を図5のように扱っている．サーバ内にはタプル以外に，ノードエントリとタプルに対するリンクが存在する．ノードエントリは，どのノードがアクティビティを実行しているかを管理する．図5で，Root から伸びているリンクをグローバルリンク，ノードエントリから伸びているリンクをノードリンクと呼ぶ．グローバルリンクは，通常のアクティビティの実行時に in と read の対象となるタプルを示す．通常時の in ではこのリンクを切ることで，タプルを見かけ上タプルスペースから削除している．図5では，in によって切られたリンクを，Root からの点線で表現している．ノードリンクは，どのノードがタスク用タプルを取り出し，アクティビティを実行しているかを示す．図5ではノード1が取り出したことを示している．

サーバ内では履歴をタスク用タプルから伸びるリンクとして実現している．履歴の各要素には，リクエストとタプルへのリンクがある*．

履歴の要素からのリンクはそのリクエストが扱ったタプルを示している．

各リクエストに対して，サーバは履歴を次のように生成する．

out タプルを生成しタプルへのリンクを張る．**out** を履歴に追加する．このとき，タプルへのリンクは NULL ポインタである．

in タプルへのグローバルリンクを切り，**in** を履歴に追加し，そこからタプルへリンクを張る．

read **read** を履歴に追加し，そこから読み込まれたタプルへリンクを張る．

message-in タスク用タプルへのグローバルリンクを切り，ノードリンクを張る．

send 履歴に **send** を追加する．タスク用タプルへのリンクは NULL ポインタである．

return 履歴に **return** を追加する．タスク用タプルへのリンクは NULL ポインタである．

receive タプルへのグローバルリンクを切り，履歴に **receive** を追加し，そこから，タプルへリンクを張る．

リクエスト **out** などに対して，履歴の要素からのリンクに NULL ポインタを用いているのは，再実行時に故障前と同じタプルを扱うからである．

アクティビティが正常に終了すれば履歴を削除する．すなわち，タスク用タプルからのリンクをたどり，履歴の各要素を削除する．次に，ノードエントリからタスク用タプルへのリンクを切り，タスク用タプルを削除する．このとき，履歴の各要素からタプルへのリンクを切る．計算用タプルは自分へのリンクの数（履歴からのリンクとグローバルリンクの和）が0になったときにタプルスペースから削除される．タプルスペースサーバはノード間での一貫性を保つので，削除されたタプルはすべてのノードのタプルスペースサーバ内のタプルスペースから削除される．

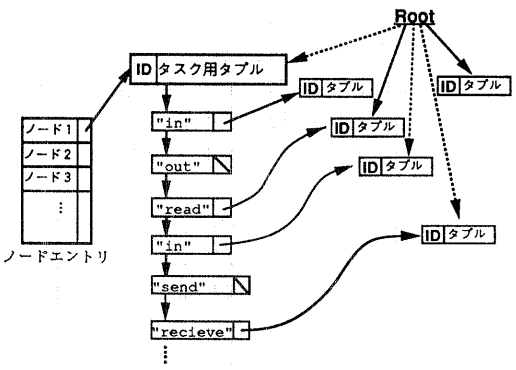


図 5 タプルの実現
Fig. 5 Implementation of a tuple.

* タプルに対する操作名はコンテキストから判別できるので，実際には必要ないが，説明を簡易にするために残している．

4.3 履歴によるアクティビティの再実行

故障を発見したサーバは他のすべてのサーバに故障を知らせる．知らせを受けたサーバはリクエストの処理を一時中断し，動かなくなったアクティビティに対応するタスク用タプルにグローバルリンクを張り直す．この張り直しによって，メッセージマネージャが，そのタスク用タプルを取り出せるようになるので，アクティビティの再実行が可能になる．リンクの張り直しが終了すれば，中断していた処理を再開する．そして，リンクを張り直されたタスク用タプルを取り出したメッセージマネージャが再実行のアクティビティを生成する．2.3 節で述べたように，メッセージは待たされることなく処理されるので，リンクを張り直せば，必ず再実行のアクティビティが生成される．これは，オブジェクト内のすべてのアクティビティの実行がブロックされているときについても保証される．

再実行のアクティビティからのリクエストをサーバは次のように処理する．

out 即座にアクティビティに操作 out の終了を知らせる.

in 履歴を使ってタプルへのリンクをたどり，故障前に取り出したタプルをアクティビティに返す.

read in と同様に履歴を使ってタプルをアクティビティに返す.

send と **return** は **out** の変形なので **out** と同じ処理をする. **receive** は **in** の変形なので **in** と同じ処理をする.

再実行時には，サーバは履歴を使ってリクエストの処理をするので，タプルスペースに変更を加えることは決してない. このために，アクティビティの再実行は通常のアクティビティの計算に影響を与えず，再実行によって他のアクティビティのロールバックは発生しない.

4.4 ノード間での一貫性の実現

各ノードのサーバは履歴を含めたタプルスペースの一貫性を保持しなければならない. 本節では Xu の耐故障性を持つタプルスペース[11]に対して，履歴の操作を追加した方式を用いる. アクティビティからのリクエスト **out**, **in**, **read** は次のように実現する.

（1） **out**

タプルと out を実行したアクティビティの ID をすべてのサーバに放送する. 受け取ったサーバはタプルをタプルスペースに加え，履歴の要素を新たに付け加える. アクティビティに作業の終了を知らせる.

（2） **in**

最初にローカルなタプルスペースでマッチするタプルを探す. そのようなタプルがあれば，そのタプルをロックし，他のノードのタプルにロックをかける. これはタプル ID とアクティビティ ID を送ることによって行う.

受け取ったサーバは，そのタプルがすでにロックをかけられていなければ，ロックし，ロックしたことを知らせる. すでにロックをかけられていた場合には，ロックを拒否する.

送り手のサーバはすべてのサーバがロックをしたことを確認した後に，タプルを取り除く指令を送る. タプルを取り除く指令を受け取ったサーバは，履歴を作成し，履歴からタプルへのリンクを張り，グローバルリンクの切断を順に行った後に，作業が完了したことを送り手のサーバに知らせる. 送り手のサーバはすべてのサーバから返事が返ってくればアクティビティにタプルの値を返す.

ロックを拒否してきたサーバがあった場合は，すべてのサーバからの返事を待ち，ロックに成功したサーバの数が全サーバ数の過半数を越えていれば，拒否したサーバに再度ロックの指令を送り，すべてのサーバがロックに成功するまで待つ. そうでない場合には，ロックを解除する指令を送る. そして，それから適当な時間を待ってから，マッチするタプルを探す段階をやり直す.

（3） **read**

ローカルなタプルスペースでマッチするタプルを探す. そのようなタプルがあれば，履歴を作成し，履歴からタプルへのリンクを張り，アクティビティにタプルを返す. このとき，次に処理する out あるいは in まで，他のサーバに履歴を伝えない. これはノード間の通信を減らすためである. マッチするタプルがない場合はマッチするタプルが現れるまで待つ.

他のリクエストは **out**, **in** の変形であるので，上述の **out** と **in** の実現を使うことで，それらのリクエストに対する一貫性を保つことができる.

われわれが実現に用いている放送はアトミックなので，あるサーバがタプルスペースの変更を放送すると，正常に動いているすべてのサーバは，これを受け取る. そして，その内容を変更するので，サーバ間の内容は同一に保たれる. サーバ間のコミットメント制御は，このアトミックな放送の実現で用いている[4].

本節で述べた read に関する履歴の扱い方では，read を実行したノードには履歴が残っているが，それ以外のノードでは履歴が残っていない状態が起こりうる. この状態で，read を実行したノードに故障が発生すると，他のノードには履歴が残っていないために，再実行時に，故障前と違うタプルを読み込む可能性がある. しかし，この場合に限り，再実行時に違うタプルを読み込んでも問題とはならない. その理由は，read によって読み込んだ結果は，そのアクティビティがタプルスペースに変更を加えるまで，すなわち，次に out または in を実行するまで，他のアクティビティには影響を与えることがなく，再実行時にはどのようなタプルを read で読み込んできても，矛盾が起きないからである.

5. 評　　価

本章では履歴を残すことがシステム全体の大きなコストにならないことを示す. 一般に分散計算では通信のコストがシステム全体のコストの大部分を占め

る[12]．このために既存の分散システムでは，通信量を減らすための工夫がなされている．LGO では3章で述べたように，故障回復のために履歴を残している．この履歴はすべてのノードに残しておく必要があり，履歴を残さない場合に比べて，通信量が増える可能性がある．しかしながら，4章で述べたように，履歴を分散タプルスペースの一貫性管理と共に残すので，大幅な通信量の増加はないと考えられる．本章でこのことを LAN 上での実験から検証する．

実験には3台の UNIX ワークステーションを使用した．それぞれのワークステーションで，履歴を取る場合と取らない場合の通信回数と実行時間を測定した．実行時間は UNIX のシステムコール gettimeofdate() を用いて測定した．測定の対象として，次の二つのアクティビティを用いた．

アクティビティ1　out と read を交互に繰り返す．繰り返しは100回．

アクティビティ2　out と in を交互に繰り返す．繰り返しは 50 回．

どちらのアクティビティもタプルの操作しか行っていないが，評価測定には十分である．なぜなら，メッセージは3.1節で述べたようにタプルを使って実現されるからである．

アクティビティ1の測定結果を**図6**（通信回数）と**図7**（実行時間）に，アクティビティ2の測定結果を**図8**（通信回数）と**図9**（実行時間）に示す．これらのグラフは，測定をそれぞれ 30 回行い，その結果をプロットしたものである．なお，個々の通信回数に変動がみられるのはタプルを取り出すとき（タスク用タプルを含む）に競合が起こり，ロックに失敗したものが，ロックを解除し，再度タプルをロックするためにノード間の通信が増えるためである．

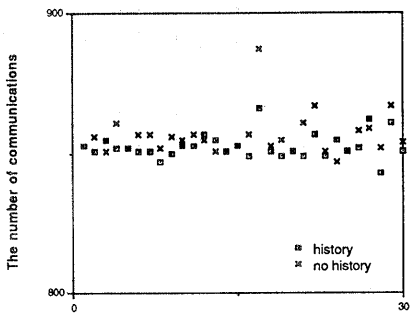


図6 結果1（通信回数）
Fig. 6 Result 1, the number of communications.

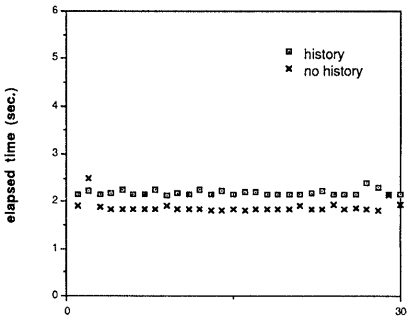


図7 結果1，実行時間（単位：秒）
Fig. 7 Result 1, the elapsed time (second).

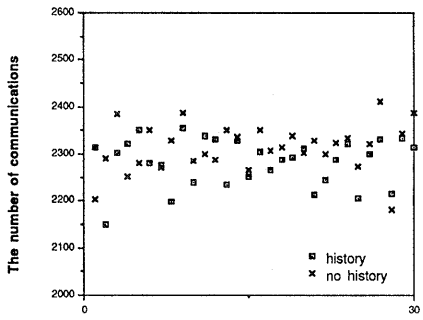


図8 結果2，通信回数（単位：回）
Fig. 8 Result 2, the number of communications.

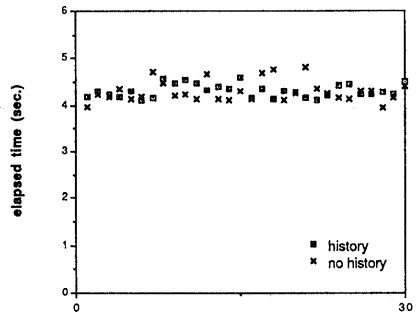


図9 結果2，実行時間（単位：秒）
Fig. 9 Result 2, the elapsed time(second).

通信回数については，どちらのアクティビティの場合も，履歴の有無に関しての差は見られない．out, in ではタプルスペースに変更を与え，その変更結果をすべてのノードに伝達する必要があり，5章で述べたように，out と in に関する履歴はこの通信と同時に伝えている．この際の通信は，履歴の有無に関わらず，必要な通信であるので，履歴を残すことによって，余分な通信は起きない．read に関しては 4.4 節で述べ

たように，次に現れる out または in まで，履歴を伝搬しない．このために out，in と同様に余分な通信は起こらない．

実行時間に関しては，アクティビティ 2 (out と in) では差が見られないが，アクティビティ 1 (out と read) では差が見られる．out と read での実行時間の差は，通信回数に差が見られないことから，履歴を残すために必要な計算時間であると考えられる．out と in で差が見られなかったのは，複数の in が競合する場合に，実行時間が遅れてしまい，その遅れが，履歴を残す計算時間より大きいためであると考えられる．

以上の結果から，履歴を残すための計算時間は余分にかかるが，通信量が大きく増えることはないといえる．さらに，履歴を残すための計算時間も，タプルのマッチングの際に起こるプロセスのブロックに比べれば，大きなものではないといえる．

6. まとめ

並列オブジェクト指向言語 LGO を用いた．分散環境における Linda のプロセスの故障回復方法について述べた．われわれの方法では，プログラマは故障の回復についての記述をする必要がないので，分散システムの構築を容易にすることができる．LGO では故障からの回復は，動かなくなったアクティビティを，オブジェクトが別のノードで自動的に動かすことによって行う．アクティビティは Linda のワーカプロセスが実行しているタスクに相当し，寿命が短いので，再実行に要する時間を縮めるためにアクティビティの状態を残す必要がない．故障回復のために再実行されるアクティビティは，操作の履歴を使うことで，故障前と全く同じ実行をする．故障前と全く同じ実行をするので，他のアクティビティのローカルバックを引き起こすことはない．そして，この履歴はタプルスペースの一貫性管理に必要な通信と同時に行われるので，履歴を残すことによるノード間の通信の大きな増加はない．

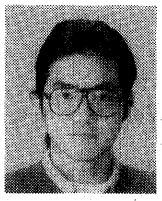
今後の課題として，LGO のコンパイラの実現があげられる．LGO のコンパイラを用いることで，プログラムは故障回復について意識することなく，より容易に分散システムを構築することができる．

謝辞 本研究の機会を与えて頂いたシステム基礎研究部武田捷一部長に感謝します．

参考文献

1) Ahuja, S., Carriero, N. and Gelernter, D.: Linda and Friends, *Computer*, Vol. 19, No. 8, pp. 26-34 (1986).
2) Bal, H. E., Steiner, J. G. and Tanenbaum, A. S.: Programming Languages for Distributed Computing Systems, *ACM Computing Survey*, Vol. 21, No. 3, pp. 261-322 (1989).
3) Birman, K. P., Joseph, T. A., Raeuchle, T. and Abbadi, A. E.: Implementing Fault-tolerant Distributed Objects, *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 6, pp. 502-508 (1985).
4) Birman, K. P. and Joseph, T. A.: Reliable Communication in the Presence of Failures, *ACM Transactions on Computer Systems*, Vol. 5, No. 1, pp. 47-76 (1987).
5) Borg, A., Baumbach, J. and Glazer, S.: A Message System Supporting Fault Tolerance, *Operating System Review*, Vol. 17, No. 5, pp. 90-99 (1983).
6) Carriero, N. and Gelernter, D.: How to Write Parallel Programs: A Guide to the Perplexed, *ACM Computing Survey*, Vol. 21, No. 3, pp. 323-357 (1989) (寺田 実訳：並列プログラムはいかに書くか：悩める人へのガイド，bit 別冊，コンピュータサイエンス，共立出版 (1991))．
7) Corradi, A. and Leonardi, L.: Parallelism in Object-Oriented Programming Languages, *International Conference on Computer Languages*, pp. 271-280 (1990).
8) Jellinghaus, R.: Eiffel Linca: An Object-Oriented Linda Dialect, *ACM SIGPLAN Notice*, Vol. 25, No. 12, pp. 70-84 (1990).
9) Stevens, W. R.: *UNIX Network Programming*, Prentice Hall, p. 772 (1990).
10) Strom, R. E. and Yemini, S.: Optimistic Recovery in Distributed Systems, *ACM Transactions on Computer Systems*, Vol. 3, No. 3, pp. 204-226 (1985).
11) Xu, A. S.: A Fault-Tolerant Network Kernel for Linda, Technical Report 424, MIT/LCS, p. 89 (1988).
12) 上林弥彦：分散技術の基本的課題，情報処理，Vol. 28, No. 4, pp. 377-386 (1987).
13) 浜田 喬：分散処理言語，情報処理，Vol. 28, No. 4, pp. 456-462 (1987).
14) 明石 修：Nuelinda における TS の制御とクラスタ化，日本ソフトウェア科学会第 8 回全国大会，pp. 201-204 (1991).

(平成 5 年 2 月 8 日受付)
(平成 6 年 2 月 17 日採録)



野里　貴仁（正会員）

昭和 63 年東京工業大学理学部情報科学科卒業．平成 2 年同大学院理工学研究科情報科学専攻修士課程修了．同年 4 月三菱電機(株)入社．中央研究所にて，オブジェクト指向言語，分散システムなどの研究に従事．日本ソフトウェア科学会会員．

杉本　　明（正会員）

昭和 52 年京都大学理学部卒業．昭和 54 年同大学院工学研究科（数理工学）修士課程修了．同年三菱電機(株)入社．以来同社中央研究所にて，設計支援システム，オブジェクト指向言語，分散システムなどの研究に従事．工学博士．電子情報通信学会，ACM 各会員．

阿部　　茂（正会員）

昭和 46 年東京大学工学部電子工学科卒業．昭和 51 年同大学院工学系研究科(電気工学)博士課程修了．工学博士．同年 4 月三菱電機(株)入社．中央研究所にて，電力系統システム，オブジェクト指向システム，幾何情報システムの研究に従事．現在，産業システム研究所勤務．昭和 60 年電気学会論文賞授賞．IEEE，電気学会，電子情報通信学会各会員．