知的分散オペレーティング・システム

正 員 田 村 信 介 (東芝)
非会員 岡 宅 泰 邦 (東芝)
非会員 関 俊 文 (東芝)

1. ま え が き

ユーザ・ニーズの多様化に伴って、高い拡張性と適応性、信頼性をもつシステムが求められている。また、ますます大規模、複雑化するシステムを、従来よりも短かい期間で開発しなければならなくなっている。システムの分散化はこれらの要求に応える有力な方法であり、多くの分散アーキテクチャが提案されている⁽¹⁾。知的分散システムもその一つであり、集中管理機構をシステムから完全に排除することにより、上述のような性質を実現している⁽²⁾⁽³⁾。

本論文では、知的分散システムを実現する基盤である知的分散オペレーティング・システム (OS) の構造と機能について述べる。知的分散 OS は、複数の計算機上に分散したプログラム群の動作を統合管理するものであり、既に提案されている多くの分散 OS と同様に、システムの分散構造に依存しないアプリケーション・プログラムの開発を可能にするが^{(4)~(7)}、これらの OS と比較して次のような特徴をもつ。

(1) OS の主要機能がアプリケーション・プログラム群の結合によって実現される。

(2) 前項に伴い、すべてのグローバルなシステム管理機能が分散アルゴリズムによって実現される。

既に多くの分散 OS においても OS の主要機能はカーネルの外へ移行しているが、システム管理の個々の機能はまだ特定の集中機構が実現しており、管理機能の分散は十分ではない。知的分散 OS はこれらの機能をすべてアプリケーション・プログラム群の結合によって行う。従って、従来の分散 OS が実現したシステ

ム分散構造からのアプリケーション・プログラムの独立性が OS のレベルまで拡大され、分散システムの高い拡張性と適応性、信頼性の実現が可能になる。

2. 知的分散システムのアーキテクチャ⁽³⁾

知的分散 OS に求められる構造と機能を明確にするため、まず知的分散システムの概要を述べる。知的分散システムは、特定の機能と性能をもつシステム要素の単なる集合から成る。これらの要素群を管理する集中機構は存在せず、状況に応じて個々の要素自身がその役割りを発見する。

これに対して従来の多くの分散システムには、システム全体の動作を集中的に管理する機構が存在し、システム要素群の結合形態や動作順序は、この集中機構が規定する範囲に束縛されていた。従って、従来形の分散システムを手続き形と呼ぶと、知的分散システムは宣言形システムと呼ぶことができる。これは従来形の手順の固定されたプログラムを手続き形、プロダクション・システムに代表される非決定的なプログラムを宣言形と呼ぶのに対応している。手続き形のシステムには、個々の要素の能力にかかわらずシステムが集中管理部の想定していない状況に対応できないなどの問題があるが、宣言形システムでは個々の要素の役割りは要素自身が決定するので、各要素がその能力を最大限に発揮できるようになり、システムの適応性が大幅に向上する。宣言形システムを実現するには、次に示す二つの機構が必要である。

(1) 個々のシステム要素が他の要素の能力に依存しないで自律的に動作する機構。

(2) 自律的に動作する要素群が、システム全体の目的を達成するために互いに協力、協調する機構。

ここで、協力は複数の要素群が同一の目的達成のために結合して互いに役割りを分担すること、また協調はシステム全体が矛盾なく、かつ効率的に動作するように異なる目的間の競合を調整することであり、いず

Operating System of the Intellectual Distributed Processing System. By Shinsuke Tamura, Member, Yasukuni Okataku, Non-member & Toshibumi Seki, Non-member (Systems & Software Engineering Laboratory, Toshiba Corporation).

田村信介：正員，(株)東芝システム・ソフトウェア技術研究所
岡宅泰邦：非会員，(株)東芝システム・ソフトウェア技術研究所
関 俊文：非会員，(株)東芝システム・ソフトウェア技術研究所

れも要素間の情報交換によって実現される。知的分散 OS が他の分散 OS と異なる点は、この協力と協調の実現を容易にするための OS 構造と機能にある。

3. 知的分散 OS の構造

知的分散 OS の管理対象は図 1 に示すように、ローカルエリア・ネットワーク (LAN) で接続された計算機上のプログラムである。前章(1)項で示した自律動作可能なシステム要素を実現するため、本 OS ではオブジェクト・モデルを採用し、個々のシステム要素に対応するアプリケーション・プログラムはすべてオブジェクトとして実現される。オブジェクトとは動作に必要なデータと手続きを自身に備えた能動的な実体であり、アプリケーション・プログラムの構成単位をオブジェクトとすることにより、自律動作可能なシステム要素実現が容易になる。

図 1 中の各計算機には、同一の機能とインタフェースをもった知的分散 OS 核が搭載されるが、これらは個々の計算機のプロセッサやメモリの管理、あるいはオブジェクト間で交換されるメッセージの送受信処理などローカルな資源管理を行うもので、各オブジェクトへの役割割り割当てなどオブジェクトのグローバルな動作管理は行わない。オブジェクトのグローバルな動作管理は、関連するオブジェクト群が結合することによりオブジェクト自身が行う。本 OS の最大の特徴は、このようにグローバルな管理機構がアプリケーション・プログラムに相当する個々のオブジェクトに分散していることであり、そのために負荷分散や同時実行制御、故障管理などの分散アルゴリズムを実装したことである。他の多くの分散 OS と同じく、本 OS もシステム分散構造の透過性を高めるが^{(4)~(7)}、更に本 OS のこの特徴は前章(2)項で示した自律動作する要素群の協力と協調機構の実現を容易にし、要素の追加や変更、移動、削除が容易な、より柔軟なシステム構築を可能にする。

図 2 に従来の分散 OS の構造を、また図 3 に知的分散 OS の構造を示す。既に従来の分散 OS においても

OS の主要機能は OS 核からオブジェクトに移行しており、OS 全体は非常に柔軟になっている。例えば OS 機能の追加や変更は、対応するオブジェクトを追加あるいは変更することにより、アプリケーション・プログラムを追加、変更するのと同じように実施することができる。しかしこの構造では、OS の諸機能がオブジェクトで実行されるものの、個々の管理機能は依然として特定のオブジェクトによって実行される。従って、システムの特典部分に負荷が集中すると共に、個々のアプリケーション・オブジェクトごとに特別の OS 機能を提供するのは困難である。知的分散システムでは各オブジェクトが固有の判断で自律的に動作するので、OS が個々のオブジェクトごとに特別なサービスを提供する能力は特に重要である。図 2 においても、もちろん個々のオブジェクトが要求する機能に応じて OS レベルのオブジェクトを生成することは可能であるが、アプリケーションと OS レベルのオブジェクトの対応関係の保守管理が繁雑になる。

これに対して図 3 では、個々のオブジェクトが自身の要求する OS 機能をすべて自身にもつので、オブジェクトに固有の OS 機能の追加や変更に必要な処理が局所化され、オブジェクトと要求する OS 機能の対応関係の管理が不要になる。従って、オブジェクトの計

図 2 従来形分散 OS の構造
Fig. 2. A conventional distributed OS architecture.

図 3 知的分散 OS の構造
Fig. 3. IDPS-OS architecture.

図 1 知的分散システムの構成
Fig. 1. IDPS configuration.

算機間移動なども他の部分から完全に独立に実施できる。図2の場合はアプリケーション・オブジェクトと、それに必要なOSレベルのオブジェクトを連動して移動しないと、処理効率低下などの問題が生じる。

図3の構造のもう一つの利点は、OS機能のマルチスレッド化が可能になることである。図2では、一つのOS機能を同時に複数のオブジェクトが要求した場合、サービスを実施するオブジェクトは同時には一つの要求にしか答えられないので、サービスを待たされるオブジェクトが生じる。従って、サービスを要求しているオブジェクト群が各々異なる計算機に搭載されているような場合には多くの計算機がサービス待ち状態になる恐れがあり、図1のような複数の計算機から成るシステムの効率的な利用はできない。図3では、オブジェクト自身が必要なOS機能をもつのでサービス要求はただちに受け付けられる。複数のオブジェクトが同一のOS機能をもつ場合、これはOS機能を実行するプログラムが複数存在してそれらが同時並列に走行することに相当する。ただし、単一の計算機に同一の機能をもつプログラムが複数個存在するのは効率が悪いので、実際には再入可能な一つのプログラムが複数のオブジェクトで共有される。OS機能のマルチスレッド化はSprite⁽⁷⁾などでも既に実現されているが、図3の構造によればOS機能のマルチスレッド化を、特別な機構を設けることなく知的分散OS核が通常のタスクスイッチを行う過程でそのまま実現できる。図2の構造の場合には、サービスを提供するオブジェクトが受け付けた要求を時分割で処理する機構などが必要になりオーバーヘッドが増加する。

各オブジェクトは、このように自身に固有な機能に相当するローカルな機構と、従来のOS機能に相当するグローバルな機構を併せもつが、知的分散OSのオブジェクト定義機能はこれらを別々に管理する。グローバルな機構を共通知識、ローカルな機構を固有知識として別管理することにより、処理の流れの把握やプログラム再利用が容易になる。ただし、処理実行時には二つの部分は区別されることなく同等に扱われる。これらオブジェクト間のメッセージ交換は放送通信による動的結合によって行われるので、各オブジェクトおよびOS核が結合相手の位置などを知る必要はない。図3の構造とこの動的結合機能によって個々のオブジェクトの独立性が更に高まり、システム運転中の要素追加、移動、変更、削除も容易になる。

4. 分散管理アルゴリズム

前章では、個々のオブジェクトにOSの機能を分散

させる構造を提案した。しかしこのような構造が動作するには、同時並列に実行される処理間で矛盾が生じないようにその進行を制御したり（同時実行制御）、個々の資源の負荷が均等になるように処理負荷を配分する、あるいはオブジェクトの故障の影響が他に波及するのを防ぐなどの機構を実現する各種分散アルゴリズムが必要である。更に、これら分散アルゴリズムは不特定多数のオブジェクト群の結合動作が前提になるので、知的分散OS核は非同期放送形の通信をサポートするが、放送によって無関係なオブジェクトに割込みがかかり、処理効率が低下するのを防ぐには、放送範囲の限定が可能な通信機能が必要になる。

本章ではこれらのうち、知的分散OSで用いる同時実行制御と故障管理の分散アルゴリズムについて述べる。なお、負荷分散およびオブジェクト間の範囲限定放送機能については既に報告したのでここでは省略する⁽²⁾⁽³⁾。

〈4・1〉 同時実行制御 同時並列に進行する処理群がデータ資源などに無秩序にアクセスすると、ある処理が参照中のデータを他の処理が変更したりして処理結果に矛盾が発生する。複数の処理群を矛盾なく進行させるには、処理のアクセス順序が資源間で逆転しないようにしなければならない。例えば、処理AとBがともに資源1と2にアクセスする場合、資源1が処理A、Bの順でアクセスされるなら、資源2でも処理AのアクセスはBに先行しなければならない。このような目的での処理の資源アクセス動作の制御は同時実行制御と呼ばれ、二相ロック法や時刻印法が広く使われているが⁽⁸⁾、知的分散OSでは並列実行処理能力に優れた二相ロック法を採用している。

本節では、二相ロック法を用いる際に発生するデッドロックの検出と防止の分散アルゴリズムを提案する。デッドロックの対処法には、資源へのアクセスをデッドロック発生の危険がなくなるまで認めない防止法と、資源へのアクセスを無条件に認めてデッドロック発生後にそれを検出し、関連する処理の中止によってデッドロックから脱出する検出法とがある。従来、二つの方法は別々に使われていたが、ここでは更にこれらの方法を併用可能なロック規約を提案する。防止法を用いるには、処理はその実行に先立って使用する資源を予約しなければならない（ロック予約と呼ぶ）が、防止法では絶対にデッドロックを発生することがなく、デッドロックのため処理を中止、再実行する必要がなくなる。従って、データベースの検索のように使用する資源が処理の進行と共に判明し、事前に確定できない場合には検出法が有効であり、外部への物理

的な出力を伴い、処理の中止や再実行が困難な場合には防止法が有効である。提案する規約により、これら二つの方法が自由に使い分けられるようになる。

デッドロック発生、あるいはその危険性の検出は、有向グラフ上の閉路検出に帰着できることが知られている⁽⁹⁾。すなわち、デッドロックである必要十分条件は、図4に示す処理による資源ロックとロック許可待ちの関係を表す Hold-Wait グラフ上に、有向枝の閉路が存在することである。また処理群が将来どのような順でロック要求を出しても、デッドロックを起こさずにすべての処理を終了する方法が存在する必要十分条件は、図5に示す処理による資源ロックとロック予約の関係を表す Hold-Claim グラフ上に、有向枝の閉路が存在しないことである。

既に、有向グラフの閉路検出分散アルゴリズムとしてプローブ法⁽¹⁰⁾や可到達集合法⁽¹¹⁾が提案されている。プローブ法では、プローブと呼ばれるメッセージをグラフの有向枝に沿って順次ノードからノードへ転送し、それがプローブ送出済みのノードに戻ったときに閉路を検出する。従って、閉路検出操作が直列的に進み並列処理の効果が期待できず、また通信エラーによるプローブ紛失などからの回復操作が複雑になる。そこで知的分散 OS では、個々のノードがそこから有向枝をたどって到達可能なすべてのノード（可到達集合）を記憶する可到達集合法を用いる。可到達集合法では、資源のロックあるいはロック予約によって有向

枝が追加されたとき、追加有向枝の根元ノードが行先ノードの可到達集合に含まれる場合に閉路を検出したことになる。

この方法によれば、プローブが順次転送されるようなことがないので並列処理の可能性があるが、処理の資源ロックやアンロック、ロック予約に伴う有向枝の追加と削除に応じて各ノードの可到達集合を変更しなければならない。しかし、有向枝の削除に伴う変更は図6に示すように複雑であり、従来は有向枝削除時の有効な可到達集合更新方法がなかったため、可到達集合法は二相ロック規約に従う場合のデッドロック検出にしか利用されなかった（デッドロック検出法では、デッドロック検出時以外に有向枝が削除されるのは処理が資源をアンロックする場合だけであるが、二相ロック規約に従えばアンロック操作を行う処理は必要なすべての資源のロックを済ませているので、図6のように削除対象有向枝の行先ノードから出る有向枝が存在することはなく、可到達集合の更新が簡単になる）。

ここでは、可到達集合法がより一般の場合にも適用できるように、効果的な可到達集合更新の分散アルゴリズムを提案する。必ずしも二相ロック規約に従わなくても処理の整合性の保証できる場合も多いが、本アルゴリズムによって可到達集合法がこのような場合にも、またデッドロックの検出だけでなく防止にも利用できるようになる。

附録に有向枝削除に伴う可到達集合の更新アルゴリズムを示す。本アルゴリズムでは、図4、図5に示した処理と資源のノード間の関係を図6のように処理ノード間だけの関係に変換した有向グラフを扱う。図6は、処理Aのロックしている資源の少なくとも一つを処理Bがロック要求あるいはロック予約しているときに、処理Bから処理Aに向う枝を定義する規則によって資源ノードを含むグラフから変換される。知的分散 OS では、個々の処理要求が発生するごとにその処理内容を表すオブジェクトが生成され、本アル

図4 Hold-Wait グラフ
Fig. 4. A Hold-Wait graph.

図5 Hold-Claim グラフ
Fig. 5. A Hold-Claim graph.

図6 有効枝削除時の可到達集合更新
Fig. 6. Maintenance of reachable sets in case of a branch deletion.

リズムはこれらオブジェクト群の共通知識上に実装される。各オブジェクトは対応するノードの可到達集合のほか、逆向可到達集合、隣接集合、逆向隣接集合を記憶しており、アルゴリズムの実行はこれらのオブジェクトがメッセージを交換することにより、各オブジェクト上で並列に進行する。

また各オブジェクトのメッセージ送信先はオブジェクト自身が把握しているので、通信エラーなどに際してもメッセージ紛失などの回復処理が容易になる。ここでノード A の隣接集合とは、 A から出る有向枝によって直接到達可能なノードの集合であり、逆向可到達集合はその可到達集合に A を含むようなノードの集合、また逆向隣接集合はその隣接集合に A を含むようなノードの集合のことである。本アルゴリズムは処理ノードの総数を n とすると、最大でも $(5n+1)$ 回の通信回数で実行できる。附録の手順(2), (4), (6)で実施されるマルチキャストを各々1回と数えると、これは $(2n+4)$ 回に減る。一つの計算機が同時に発生する処理の最大数を設定すると、計算機間で情報を交換することなく、システム中の全処理に一意に定まる番号を与えることができ、可到達集合は n に比例するビット長のパターンとして表現することができる。

従って、付録の手順で必要になる集合演算はビットパターンの論理和、論理積、排他的論理和に帰着され、個々のノードの計算量が n のオーダーに納まる。全ノードの計算量を合せると n^2 のオーダーになるが、この計算はノードが搭載される計算機群上で並列に実行することができる。総計算量に関しては、従来報告されている分散デッドロック検出、防止アルゴリズムも n^2 または n^3 のオーダーであり⁽¹²⁾、本アルゴリズムが特に優れていることはないが、他のアルゴリズムで必要になる通信回数は n^2 以上のオーダーであり、処理の並列性や通信回数を考えると実質的には本アルゴリズムのほうが優れていることがわかる。

デッドロック検出と防止は前述したように使い分けのことができるが望ましい。すなわち、アクセス中の処理の中止と再実行が許される資源をタイプ1、そうでないものをタイプ2とすると、タイプ1資源に対しては検出法を用い、タイプ2資源に対しては防止法を用いる。このとき、デッドロックのためにタイプ2資源にアクセス中の処理が中止されてはならない。

既にこのような条件を満たす方法として、資源タイプに優先度を付け、処理の資源へのロック要求は必ずそれより優先度の高いタイプの資源を、すべてアンロックした後でしか許さないロック規約が提案されてい

る⁽¹³⁾。しかし、そこでは優先度の設定に規定がないので、一つの処理が異なるタイプの資源に同時にアクセスすることは許されなかった。ここでは、資源タイプ1の優先度をタイプ2より高く設定することにより、処理が同時に異なるタイプの資源にアクセスできるロック規約を提案する。更に、各処理が資源のタイプ分類をほかと独立に行えることを示す。つまり、ある処理がタイプ1に分類した資源を別の処理がタイプ2に分類してもかまわない。次に提案ロック規約を示す。

〔規約1〕 タイプ2資源をロック中にタイプ1資源へロック要求を出してはならない。

〔規約2〕 ロック予約をしていないタイプ2資源にロック要求を出してはならない。

〔規約3〕 処理は自身がタイプ2資源をロックしていないときにのみロック予約ができる。

〔規約2〕と〔規約3〕はロック予約に関する通常の規則であり、本規約に特有な規則は〔規約1〕だけである。〔規約1〕は、タイプ2資源ロック中処理のタイプ1資源ロックを禁止するが、タイプ1資源ロック中処理のタイプ2資源ロックは禁止しない。従来の規約では、タイプ1とタイプ2への優先度の与え方に規定がないのでこの両方が禁止される。本規約下でのデッドロックの検出と防止は次のように行われる。

つまり、他の処理が資源をロック中のため、ある処理のタイプ1資源へのロック要求が待たされたとき、デッドロック検出アルゴリズムが Hold-Wait グラフの閉路検出を行う。その結果、閉路を検出するとデッドロック状態から脱出するためにロック要求処理を中止するが、そうでない場合は単に資源がアンロックされるのを待つ。処理がタイプ2資源をロックする場合には、必ずデッドロック防止アルゴリズムが動作し、Hold-Claim グラフ中の閉路検出を行う。その結果、閉路を検出すると関連資源がアンロックされて閉路がなくなるまでロック要求を待たせるが、そうでない場合はただちに資源をロックする。タイプ2資源のロックを終了した段階で更にデッドロック検出アルゴリズムが動作し、Hold-Wait グラフ中に閉路を検出すると、閉路に含まれるタイプ1資源のみをロック中の処理を中止してデッドロックから脱出する。

本ロック規約が正しく動作することは次のように容易に示すことができる。まずタイプ1資源のロックによって Hold-Wait グラフに閉路が生成された場合は、〔規約1〕によりロック要求を出した処理はタイプ1資源しかロックしていないので、それをアンロックすることにより、デッドロック状態から脱出できる。

またタイプ2資源のロックによって Hold-Wait グラフに閉路が生成される場合も、Hold-Claim グラフに閉路のないことが確認されているので、Hold-Wait グラフ上の閉路がタイプ2資源をロックしている処理だけから構成されることはなく、閉路に含まれるタイプ2資源をロックしていない処理を中止することによりデッドロック状態から脱出できる。

本規約では、処理によって資源のタイプ分類が異なることを許しているが、許さない場合は(規約1)によって Hold-Wait 関係と Hold-Claim 関係を示す有向枝の混在した閉路が生成されることはない。二つの関係の混在した閉路が形成されるには、少なくとも一つの処理がタイプ2資源ロック中にタイプ1資源にロック要求を出さなければならない。従って、資源のタイプ分類が処理間で共通な場合には、Hold-Wait グラフと Hold-Claim グラフを別々に管理する必要はなく、それらを混合したグラフを一つ管理すればよい。

〈4・2〉 故障管理 高信頼化方式には、同一の処理を複数の要素で並列に実行する並列多重処理方式と、処理は単一の要素で実行し、実行中の要素が故障した段階で予備要素が処理を継続する待機冗長処理方式とがある。待機冗長処理方式では予備要素は実際の処理を行わないので、単一の予備要素が複数の要素の予備を兼ねるなど、要素の冗長度が全体として低くなる。しかし故障時に予備要素が処理を引継ぐのに時間がかかる、あるいは高速で処理を引継ぐには高頻度で予備要素に処理の途中結果を通報しなければならないなどの欠点がある。リアルタイム制御などでは、高信頼化によるオーバーヘッドの増加や故障による休止時間も極力小さく抑える必要があるので、知的分散 OS では並列多重処理方式を採用する。このとき、システムの適応性や拡張性を保つには、個々の要素は多重化の有無あるいはその程度と独立に定義、利用できなければならない。またリアルタイム制御などで要求される処理性能を得るには、多重要素間での同期操作などを極力省略する必要がある。

図7はこれらの要求を満たすために考案されている方式で^{(14)~(16)}、個々の要素は放送通信によって多重化された要素群を同時に呼び出す。図7では、要素Aが放送通信によって多重化要素B1, B2, B3を呼び出し、更にB1~B3の各々が放送通信によって多重化要素C1, C2を呼び出す。各要素は多重化された要素群から同一内容のメッセージを受け取ると、最初に到着したメッセージのみ受け取り、他は無視するなどの手段で同一のメッセージに複数回答するのを防ぐ。この方式では、多重化要素群を放送通信で呼び出すの

図7 放送結合による並列多重処理
Fig.7. Behavior of replicated elements connected through broadcast communication.

で、関連要素の多重化の有無やその程度を知ることなく、各要素を定義、利用することができる。しかし、この方式の実現には、次に示す二つの機構が必要になる。

(1) 正しく動作している多重要素群はすべて同一のメッセージを同一の順序で受け取る。

(2) 個々の要素は正しいメッセージしか受け取らない。

(1)項の機構については二相コミット規約の応用やデッドロック検出機構を用いる方法が提案されているが^{(14)~(16)}、多重要素間で同期をとる必要があったり、多量の処理あるいは記憶を要するなど多重化のオーバーヘッドは高い。(2)項の実現には、個々の要素の完全な自己診断機能を仮定しない限りは⁽¹⁶⁾、多重化要素群の出力の多数決をとる方法しか存在しないが、従来は個々の要素が関連要素の多重度を知らなければならなかったり、多重化されたすべての要素からメッセージが届くのを待つ必要があるなどの好ましくない性質があった。ここでは、これらの欠点を除いた分散機構を提案する。

知的分散 OS では、多重化要素群は多重化されたオブジェクト群として実現され、これらは互いに異なる計算機上に搭載される。オブジェクトへの処理要求や処理に必要なデータの伝達は、図7と同様に放送通信によって多重化オブジェクト群に対して同時に行われる。正しく動作している多重要素群が、すべて同一のメッセージを同一の順序で受け取ることの保証は知的分散 OS 核の通信機能が行う。

すなわち、知的分散 OS 核は LAN を通過する放送モードのメッセージを常に監視しており(多重化の可能性のあるオブジェクトへのメッセージ伝達はすべて放送モードで行われる)、これまでに受け取った放送モードのメッセージを特徴づける量の異積量を記憶する。このとき、正常に動作している計算機群は同一の放送モードメッセージ群を受け取るの、それらの上の OS 核が記憶している異積量はすべて等しいはずで

ある。そこで各 OS 核は送信すべきメッセージに自身の記憶している累積量を付加し、受信側 OS 核がメッセージに付加された累積量と自身が記憶している累積量とを比較する。受信側 OS 核は更に、二つの累積量が異なる場合は送信側が故障したと判断して、受信メッセージを無視すると共に送信側 OS 核に対して故障通知を送る。各 OS 核は、一定数以上の計算機から故障通知を受け取ると自身が故障と判断して、その動作を停止する。ここでメッセージを特徴づける量としては、メッセージ個数、長さ、あるいはチェックサム値などを用いる。

本方式では、オブジェクトが故障などの理由で（実際にはオブジェクトの搭載された計算機の故障）メッセージを受信できなかったり余分に受信したりすると、そのオブジェクトが出すメッセージに付加される累積量は正常な計算機が記憶している累積量と異なるので、正しく動作しているオブジェクト群は同一のメッセージを受け取ることができる。また、メッセージ送受信に異常のあるオブジェクトの出力は他から無視されるので、異常の影響が他に波及することもない。多重化オブジェクト群が同一の順序でメッセージを受け取る機構は、OS 核が放送モードメッセージを LAN に正しく送出したのを確認してから、自計算機上のオブジェクトに転送することにより簡単に実現できる。本方式では、更に定められた個数以上の計算機から故障通知を受け取った計算機は停止するので、故障計算機の出すメッセージがむだに LAN を占有することもない。

このとき、瞬時的な故障であっても故障計算機の動作が止まるので、停止中に進行した処理と矛盾しないように計算機を再立上げするオーバーヘッドは高くなるが、これは LAN を多重化することによって防ぐことができる。つまり、各計算機は少なくとも一つの LAN を通じてメッセージの正しい送受を行っていれば動作を続ける。ある LAN を通じて一定数以上の故障通知を受け取った計算機はその LAN からは一たん切離すが、その累積量を他の計算機の累積量に強制的に一致させた後、再び接続する。

本方式は明らかに集中管理機構を必要としない。しかも、個々のオブジェクトおよび OS 核が多重化されたオブジェクトの存在やその数を知る必要がないので、多重化すべき要素の追加や削除が完全に局所的な操作で実現され、高信頼化のために知的分散システムの適応性や拡張性が損なわれることはない。更に二相コミットメントのような複雑なメッセージのやりとりやオブジェクト間での同期が不要であるだけでなく、

実際には必要な処理が各計算機に付加される通信用プロセッサで実行できるので、オブジェクトの多重化による処理効率の低下もほとんどない。

本方式が正常に動作する前提には、各 OS 核が累積量を正しく計算、記憶、比較する必要があるが、これについても個々の計算機の通信プロセッサを多重化することにより、他に影響を与えることなく完全に局所的な処理で任意の信頼度を実現することができる。全体の信頼度についても、システムを構成する計算機の数と故障と判定するまでの故障通知の数によって任意に設定することができる。

次に個々のオブジェクトが正しいメッセージだけを受信する機構を考える。個々のオブジェクトが完全な自己診断能力をもてば、オブジェクトは正しいメッセージしか出力しないので、この機構は不要になるが、高度な自己診断機能の実現は困難であるので、知的分散 OS では多重化されたオブジェクト群のメッセージを比較する方式を用いる。多重化されたオブジェクト群のメッセージを比較する方式では、通常は多数決などによって正しいメッセージを抽出するが、そのためには各オブジェクトが関連オブジェクトの多重度を知る必要があり、また多数決に必要な数のメッセージが届くのを待たなければならない。

そこで知的分散 OS では、各オブジェクトが多重化オブジェクト群が送信したメッセージの最初に到着したもののだけを受信し、他を無視する先着優先方式を用いる。先着優先方式は既に他のシステムでも利用されているが^{(14)~(16)}、そのままでは誤まったメッセージが受信される可能性が残るので、ここではオブジェクトが定められた数（有効個数）の内容が一致するメッセージを受信した段階でそれを有効化する方式を提案する。従って、有効個数を調整することによってメッセージ内容の信頼度を任意に設定することができる。また各オブジェクトは関連オブジェクトに要求される信頼度（メッセージの有効個数）だけを知ればよく、関連オブジェクトの多重度を知る必要はないので、拡張性や適応性が低下することもない。すべてのオブジェクトの有効個数が 2 以上の場合には、ここに示した機構だけでも多重化された各オブジェクトが同一メッセージを受信することの保証は可能であるが、実際のシステム構成では信頼度のあまり要求されないオブジェクトの多重度は、1 に設定される場合が多いので前述の機構が必要になる。一般に、異常は計算機本体よりも通信路で発生することが多いので、多重度が 1 であっても前述の放送機構だけでかなりの信頼度向上が実現できる。また故障オブジェクトが余分のメッセージ

を出力する場合などには、関連オブジェクトが比較すべきメッセージとして他の多重化オブジェクトの出力を待つ必要があり、この機構だけでは処理効率が落ちるが、前述の機構によって余分のメッセージ受信が防止されこのような待ち時間もなくなる。なお、オブジェクトが受け取るメッセージ内容の比較には、比較すべきメッセージ群を識別するためにメッセージに名前付けをする分散機構が必要になるが、これについては別に報告する。

5. 終りに

知的分散 OS の構造とそこで採用した各種分散管理アルゴリズムについて述べた。本 OS により同時実行制御や故障管理などすべての機構がアプリケーション・オブジェクトに分散され、OS 機能も含めたシステムの拡張性や適応性、信頼性の向上が可能になる。現在、オフィス向けのシステムとして、LAN で結合したパーソナル計算機群上に既存 OS のゲスト OS として動作する知的分散 OS を搭載し、そのうえに多重データベースのロック管理やデータ読み出し時の負荷分散機構を実現している。またゲスト OS 方式では標準的な OS の 3～5 倍のオーバヘッドがあったが、独立 OS 化によって通常と同等の処理効率を得ることができたので、今後リアルタイム制御などの分野への適用を進める。

(昭和62年8月25日受付、平成元年1月25日再受付)

文 献

- (1) 「分散と協調特集」、計測と制御, 26 (昭62-1)
- (2) 田村・辻: 「知的分散システム」, 電学誌, 106, 39 (昭61-3)
- (3) 田村, 他: 「知的分散システムのアーキテクチャ」, 電学論C, 108, 393 (昭63-6)
- (4) A. S. Tanenbaum, et al.: "Distributed Operating Systems", *Computing Surveys*, 17 419 (1985)
- (5) M. Accetta, et al.: "Mach: A New Kernel Foundation for UNIX Development", Proc. of USENIX 1986 Summer Conference, p. 93 (1986)
- (6) D. R. Cheriton: "The V Distributed System", *CACM*, 31 314 (1988)
- (7) J. Ousterhout, et al.: "The Sprite Network Operating System" *IEEE Computer*, 21, 23 (1988)
- (8) 上林: 「共有データベースの諸問題に対する理論」, 情報処理, 24, 1020 (昭58-8)
- (9) R. C. Holt: "Some Deadlock Properties of Computer Systems", *Computing Surveys*, 4, 179 (1972)
- (10) K. M. Chandy, et al.: "Distributed Deadlock Detection", *ACM Trans. Computer Systems*, 1, 144 (1983)
- (11) S. S. Isloor, et al.: "An Effective ON-LINE Deadlock Detection Technique for Distributed Database Management Systems", Proc. COMPSAC'78, p. 283 (1978)

- (12) E. Knapp: "Deadlock Detection in Distributed Databases", *Computing Surveys*, 19, 303 (1987)
- (13) J. H. Howard Jr.: "Mixed Solution for the Deadlock Problem", *Communications of the ACM*, 16, 427 (1973)
- (14) E. C. Cooper: "Replicated Distributed Programs", Proc. of the 10th ACM symp. on Operating System Principles, Operating System Review, vol 19, no. 5 p. 63 (1985)
- (15) D. Powell, et al.: "The Delta-4 Approach to Dependability in Open Distributed Computing Systems", Proc. on the 18th International Symposium on Fault Tolerant Computing, p. 246 (1988)
- (16) R. F. Cmelik, et al.: "Fault Tolerant Concurrent C: A Tool for writing Fault Tolerant Distributed Programs", *ibid.*, p. 56 (1988)

付 録

有向枝削除に伴う可到達集合の更新

以下の手順では、ノード X の可到達、逆向可到達、隣接、逆向隣接集合をそれぞれ $FR(X)$, $BR(X)$, $FD(X)$, $BD(X)$ と表し、ノード A から B に向う有向枝を除く際の可到達集合更新を考える。

- (1) B が A に $\{B, FR(B)\}$ を送る。
- (2) A が $FD(A)$ から B を除き、 $FR(A)$ から B と $FR(B)$ のノードを除く。 A は更に $BR(A)$ に属するノード群に $\{B, FR(B), A, BR(A)\}$ を送る。
- (3) $BR(A)$ のノード C が、 $FR(C)$ から B および $FR(B)$ のノードを除く。 C は更に $T(C)=FD(C)-BR(A)$ を計算し、 A に $\{C, T(C), BR(C)\}$ を返送する。
- (4) $BR(A)$ の全ノードから返答を受信した A が、 $\{i, BR(i)\}$ ($i \in BR(A)$) を記憶し、 $Z(A)=UT(i) \cup FD(A)$ のノード群に $\{i, T(i)\}$ ($i \in BR(A)$) の列を送る。

(5) $\{i, T(i)\}$ の列を受信した D が、 D が列中の $T(j)$ に含まれるとき、 A に $\{D, FR(D), j\}$ を返送する。

(6) 返答 $\{k, FR(k), k_j\}$ を受信した A が、 k が $FD(A)$ に含まれるときは、 $FR(A)=FR(A) \cup FR(k) \cup k$ とする。 A は更にすべての $k \in Z(A)$ から返答を受け取ると、さきに記憶した $\{i, BR(i)\}$ の列中に $i=k_j$ なるものが存在する k_j を集めて、 $\{k, FR(k), BR(k_j)\}$ の列を作り $BR(A)$ のノード群に送る。

(7) $\{k, FR(k), BR(k_j)\}$ の列を受信した E が、 E が $BR(k_j)$ に含まれるとき、 $FR(E)=FR(E) \cup FR(k) \cup k$ とする。

複数の更新要求が同時に発生する場合の本手続き実行順序の制御は時刻印法⁽⁸⁾を用いて行う。また BR , BD の更新も同様に行うことができる。