社団法人　情報処理学会　研究報告
IPSJ SIG Technical Report

2006－DPS－126（44）
2006－CSEC－32（44）
2006／3／17

分散リアルタイムシステムに向けた動的リソース管理ミドルウェアの設計

村山和宏† 大谷治之† 佐藤裕幸†
目黒正之† 宮森信之† 落合真一†

近年, センサデータ処理システムなどの分散システムにおいて, より少ない計算機リソースでハードリアルタイム処理を実現することが求められている. このようなシステムでは, 各プロセスの振る舞いを監視し, デッドラインミスが発生する前に計算機リソースの再配置を行うことが必要となる. 我々は大規模センサ処理システムに対し, デッドラインミスの発生時刻を予測し, その時刻よりも前にリソース配置の変更を完了することによって システム全体のハードリアルタイム処理継続を可能とするミドルウェアの設計を行った. 本ミドルウェアでは, デッドラインミス発生時刻の予測, リソース再配置時における演算データの継続などを実現する.

Design of dynamic resource management middleware for distributed real-time systems

Kazuhiro Murayama, Haruyuki Ohtani, Hiroyuki Sato,
Masayuki Meguro, Nobuyuki Miyamori and Shinichi Ochiai†

Dynamic real-time systems, such as sensor information processing systems, should keep QoS requirements in case of constant changes of external environments, overload of internal systems, system failure, and so on. To enable this demand, we have been developing dynamic resource management middleware which performs prediction of QoS failure, reallocation of computer resources to achieve acceptable levels of QoS, data consistency between replica processes. In this paper, we describe the design of our middleware.

1. はじめに

近年, センサデータ処理システムなどの大規模分散システムにおいて, 負荷の変動に関わらずハードリアルタイム処理を継続することが求められている. このようなシステムでは, 各プロセスが必要とする計算機リソース量を常に監視し, デッドラインミスが発生する前に計算機リソースの再配置を行わなければならない.

我々はデッドラインミスの発生時刻を予測し, その時刻よりも前に 各プロセスに割り当てるリソース量の変更を完了することにより, デッドラインミス発生を未然に防止する「動的リソース管理ミドルウェア」の設計を行った. 本稿では, ミドルウェアの設計内容について述べる.

2. 背　　景

2.1 システムの特徴

図 1 は, 本研究において対象とするセンサシステムの構成概要を示したものである. 本システムは, 数十台の計算機をネットワーク接続したクラスタ, 複数のセンサ, および出力装置によって構成される. 本センサシステムでは, 自然界に存在する物体を検出し, 複数の分散アプリケーションが物体の判別, 位置の特定などの解析処理を行っている.

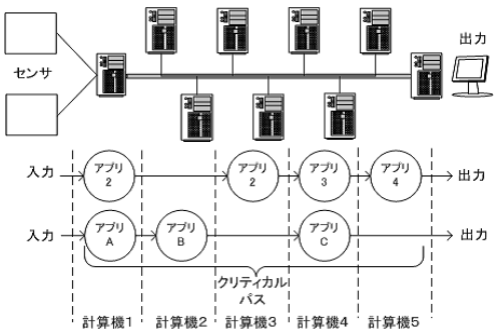

図 1　システム構成

本システムの主な特徴は以下の通りである.

- 各センサからの入力に対し, 複数個のプロセスをパイプライン状に連ねた「クリティカルパス」を処理単位とし, 複数のクリティカルパスが計算機リソースを共有して処理を行う.
- センサからは周期的にデータが送信されており, 各クリティカルパスでは次の周期までに一連の処理を完了させる「分散リアルタイム処理」が行われる.
- センサから送信されるデータの内容や大きさは, 物体の数などの外部環境によって随時変化し, それに応じてプロセスの処理量も変動する. ただし, データが変化してもクリティカルパス内の全プロセスの処理量が一律に変動するのではなく, データの内容などによって処理量が変動するプロセスは異なる.

† 三菱電機株式会社
Mitsubishi Electric Corporation

- 本システムにおいて，センシング対象となる物体は無限に出現することが考えられる．そのため，負荷の上限を予測してシステムを構築することは困難であり，演算量に対して計算機リソースが不足する可能性がある．

2.2 システム要求

本研究で対象としているシステムの要求として，主に以下のようなものがある．

ハードリアルタイム処理の継続： 前述の通り，センサは周期的にデータを送信しており，センサからのデータが増加しても，全クリティカルパスは計算機リソースに余裕がある限り次の周期までに処理を完了させなければならない．また，計算機リソースが不足している場合には，負荷の大きなクリティカルパスの処理が重要であるとし，その処理のハードリアルタイム処理を優先して継続する必要がある．

演算データの継続： クリティカルパスを構成する各アプリケーションは，過去の演算データがなければ演算を継続することができない．そのため，リソース再配置，プロセス停止，計算機故障などが発生しても過去のデータを復旧させて処理を継続することが必要となる．

2.3 関 連 研 究

これまで，高い信頼性とリアルタイム性の両立を必要とするシステムでは，負荷の上昇や計算機故障時においてもリアルタイム性を保持するするために，システム全体を冗長化し，各アプリケーションが実行される計算機をあらかじめ割り付けておく「静的リソース管理」を採用していた．しかし，本研究のターゲットのような，負荷の上限の予測が困難なシステムに静的リソース管理を採用した場合，膨大な数の計算機を用いてシステムを構築せねばならず，かつ，平常時には無駄な計算機リソースが多くなるという問題点があった．

このような問題を解決するため，近年，システム中にある余剰リソースを動的にプロセスに割り付けることによって，より少ない計算機リソースでシステムのリアルタイム処理を可能とする「動的リソース管理」技術の研究が行われている．

文献 1)，2) の研究成果である DeSiDeRaTa[1] や QARMA[2] といったリソース管理ミドルウェアでは，デッドラインミスが発生した場合に デッドラインミス発生の原因となったプロセスの負荷の軽い計算機へのマイグレーション，プロセスの分割などといったリソース割当の変更を動的に行うことにより，対処後のデッドラインミス再発防止を実現している．また，文献 3) では DeSiDeRaTa ミドルウェアを拡張し，システム全体のリソース配置の最適化を行う分散リアルタイムスケジューラを設計している．本スケジューラを任意のタイミングで動作させることによってシステムのデッドラインミス防止やスループットの向上が実現できる．

動的リソース管理ミドルウェアの代表例である DeSiDeRaTa ミドルウェアのソフトウェアコンポーネントの構成を**図 2** に示す．DeSiDeRaTa ミドルウェアは主に Software Monitor, Resource Monitor, Resource Manager, Control, Specification という 5 つのコンポーネントで構成される．各コンポーネントは以下のような機能を持つ：

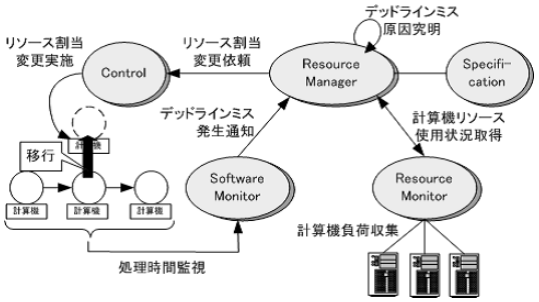

図 2 DeSiDeRaTa ミドルウェアのソフトウェアコンポーネント構成

Software Monitor： 各クリティカルパスおよびプロセスの処理時間を監視し，クリティカルパスのデッドラインミスが発生した場合には Resource Manager に警告を通知する．

Resource Monitor： 各計算機の CPU 負荷，メモリ使用状況を調査し，Resource Manager に通知する．

Resource Manager： Software Monitor からデッドラインミス発生の警告が届くと，デッドラインミス発生の原因となったプロセスを特定し，デッドラインミス発生の原因となったプロセスへの計算機リソース再配置方法（他の計算機へのマイグレーション，処理の分割など）を決定する．

Control： Resource Manager の決定に基づいてリソース再配置を行う．

Specification： システムを構成するプロセス，使用メモリ量，計算機数，各クリティカルパスのデッドライン時間など，システム構成に関する情報を保持する．

DeSiDeRaTa ミドルウェアにおける，デッドラインミス検出からリソース再割付完了までの動作の一例を示す．

(1) Software Monitor は，全クリティカルパスおよび各プロセスの処理時間を監視し，デッドラインミスが発生していた場合にはデッドラインミスの警告を Resource Manager に通知する．

(2) Resource Manager は，Software Monitor から通知を受けると，プロセスの処理時間をもとにデッドラインミス発生の原因となったプロセスを検出する．

(3) Resource Manager は，各計算機の空きリソースを調査し，デッドラインミスなく処理が可能な計算機を選択する．

(4) Resource Manager は，デッドラインミス発生の原因となったプロセスを，(3) で選択した計算機上に移行するよう Control に指示する．

(5) Control は，デッドラインミスの原因となったプロセスを停止し，(3) で選択した計算機上で再起動する．

DeSiDeRaTa ミドルウェアでは，(1) 〜 (5) の処理を短時間で行うことによりデッドラインミスの継続発生防止を実現している。

2.4 本研究の課題

本研究にて対象とするシステムに DeSiDeRaTa のようなリソース管理ミドルウェアを適用することにより，デッドラ

インミス発生の原因プロセスにリソースを動的に割り当て, リソース再配置後のリアルタイム処理継続が可能となる.

しかし, 既存の研究成果そのままでは以下のような課題があり, システム要求を満たすことが困難である.

- 頻発するデッドラインミスへの対処が困難：文献 3) では, アプリケーションの処理時間の変動とは関係なく任意のタイミングでリソース再配置を行うことにより, デッドラインミス発生を未然に防いでいる. 本手法をデッドラインミス発生頻度の高いシステムに適用した場合, リソース再配置を行っている最中にデッドラインミスが発生する可能性がある. 今回のシステムはデッドラインミス発生頻度が非常に高いことが想定され, 本手法によりデッドラインミスが解消される可能性は低い.
- 演算データの継続が不可能：関連研究では, プロセスを他の計算機に移行させた場合, 移行前のプロセスが持つデータを移行後のプロセスに受け渡す機能が実現されていない. そのため, リソース再割当を行った場合には, その時点までに行われた演算データは失われてしまう. また, 演算データのバックアップ機能を保持していないため, 計算機故障が発生した場合もデータは失われる.

文献 3) のように, リソース配置の最適化を随時行うことはシステムのリアルタイム処理の阻害につながる可能性があり, システムがハードリアルタイム処理を継続している場合はリソース管理ミドルウェアは動作せず, デッドラインミスが発生する場合にのみ動作すべきである. また, 本システムでは負荷が突発的に上昇する可能性があるため, デッドラインミスを防止するためにはリソース割当計算機の判定に要する時間を短縮することも必要になる.

以上を踏まえ, 本研究では図 2 の構成をもとに, デッドラインミス発生が予想される時にのみ動作し, ハードリアルタイム処理と演算結果の継続を可能とするリソース管理ミドルウェアの設計を行う.

3. リソース管理ミドルウェアの設計

3.1 課題の解決に向けて

システム要求実現に向けた課題の解決方法は以下の通り.

- ハードリアルタイム処理を実現するために：
  - デッドラインミス発生時刻の予測, 事前の対処完了：各プロセス, クリティカルパスの処理時間の変動をもとにデッドラインミスの発生を予測し, その時刻に先駆けてリソース再配置を実施することによりデッドラインミスの発生を未然に防ぐ. また, 将来のリソース使用状況を想定したリソース再配置を行うことにより, リソース再配置後の長期間にわたるデッドラインミス発生を防止する.
  - 突発的なデッドラインミス発生を考慮したリソース再配置の実現：突発的なデッドラインミス発生を防止するためには, リソース割当計算機選定の時間を短縮することが必要である. 本研究では, 負荷の高い計算機のリソースを優先的に使用することによって空きリソースを 1 台の計算機に集約させ, 緊急時におけるリソース割当計算機の選定を容易にする.
- 演算結果を継続させるために · · · 待機プロセスによるデータのバックアップ：リソース再配置前後で演算結果を継続させるために, あらかじめ待機プロセスを起動しておき, 待機プロセスが演算結果をバックアップする. また, 待機プロセスの配置を適切に行うことにより, システムのハードリアルタイム性向上も実現する.

以下, 設計内容について述べる.

3.2 設 計 内 容

3.2.1 デッドラインミス発生前の事前対処の実現

本研究では, 以下の (1) 〜 (4) の手法によりデッドラインミス発生時刻を予測し, その時刻より前にリソース再配置を完了する (**図 3**).

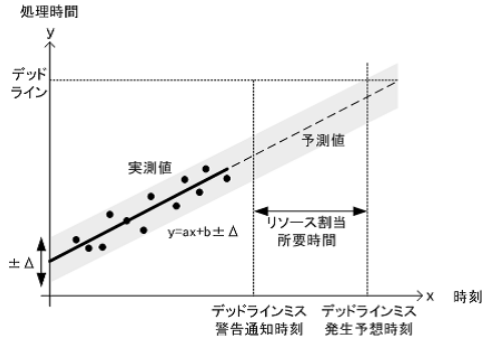

図 3 デッドラインミス発生時刻の予測

(1) 処理時間の近似式の作成： リソース管理ミドルウェア内部で, 直近 n 周期のクリティカルパスおよび各プロセスの処理時間の履歴を保持する. そして, この履歴データをもとに, 最小二乗法などによって処理時間の変化の近似式 ($y = ax + b \pm \Delta$) を求める.

(2) デッドラインミス発生予定時刻の算出： クリティカルパスの処理時間が増加傾向である場合は, (1) で求めた近似式の左辺 (y) に, あらかじめ定められているクリティカルパスのデッドラインの値を代入し, このときの x の値を求める. 得られた値がデッドラインミス発生予定時刻となる.

(3) リソース割当所要時間より前のデッドラインミス警告通知： (2) で求めたデッドラインミス発生予定時刻と現在の時刻を比較し, デッドラインミス発生までの残り時間がリソース割当計算機を求めるのに要する時間に近い場合にデッドラインミス解消に向けた対処を開始する.

(4) 将来のリソース使用状況に基づくリソース再配置： デッドラインミス発生予定時刻におけるデッドラインミス発生を防止するため, 現状のリソース使用状況ではなく, (2) で取得したデッドラインミス発生予定時刻での各プロセスのリソース使用状況をもとにリソース割当を行う計算機を決定する.

3.2.2 突発的なデッドラインミス発生を考慮したリソース再配置の実現

本節では, 3.2.1節の (4) で行うリソース再配置の方法に

ついて説明する．本研究では，リソース再配置は以下の手順にて行う．

- 手順１：現状の空きリソースを用いたリソース再配置
- 手順２：システム内に空きリソースがない場合，空きリソースの作成による重要処理へのリソース再配置

以下、これらの再配置方法について説明する。

手順１：現状の空きリソースを用いたリソース再配置：

手順 1-1： 十分な空きリソースを持つ計算機の調査： まず，デッドラインミス発生の原因プロセスが制限時間内に動作可能なリソースを持つ計算機の有無を調査する．
一般に，以下の２つの条件を満たせば，その計算機はリソース割当計算機の候補となる．

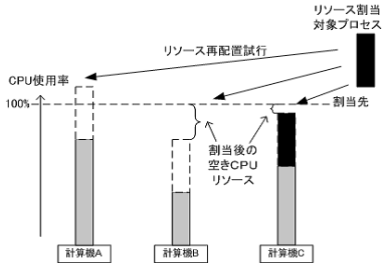

図 4 手順 1-1. の操作

(a) リソース割当対象プロセスが制限時間内に処理が完了する．

(b) (a) を満たす計算機上で動作する，他の全てのプロセスが制限時間内に処理を完了する．

条件 (a), (b) の判定方法は以下の通りである (**図 5**)：

- (a) の判定方法：リソース割当対象プロセスを t_x，プロセス t_x が属するクリティカルパス CP のプロセス群を $CP = \{t_1, \cdots, t_{x-1}, t_x, t_{x+1}, \cdots, t_n\}$，プロセス $t_i (\in CP)$ の処理時間 (処理開始から処理終了までに要した時間) を $Exec(t_i)$，クリティカルパス CP の制限時間を $DL(CP)$ とする．
  プロセス t_x に与えられる制限時間 $DL(t_x)$ は以下の式で求められる．

$$DL(t_x) = DL(CP) - \sum_{i=1}^{x-1} Exec(t_i) - \sum_{i=x}^{n} Exec(t_i) \quad (1)$$

  さらに，計算機 A で動作するプロセスおよび t_x を優先度の低い順に並べたものを，$T_A = \{t_1, \cdots, t_l, t_x, t_{l+1}, \cdots, t_m\}$，プロセス $t_i (\in T_A)$ の CPU 時間 (実際に CPU リソースを占有した時間) を $Cpu(t_i)$，プロセス t_i の処理周期を $C(t_i)$ とする．以下の式を満たせば，リソース割付対象プロセス t_x は制限時間内に処理が可能となる．

$$DL(t_x) - \lceil \sum_{i=l+1}^{m} \frac{DL(t_x)}{C(t_i)} \rceil \times Cpu(t_i) - Cpu(t_x) > \Delta \quad (2)$$

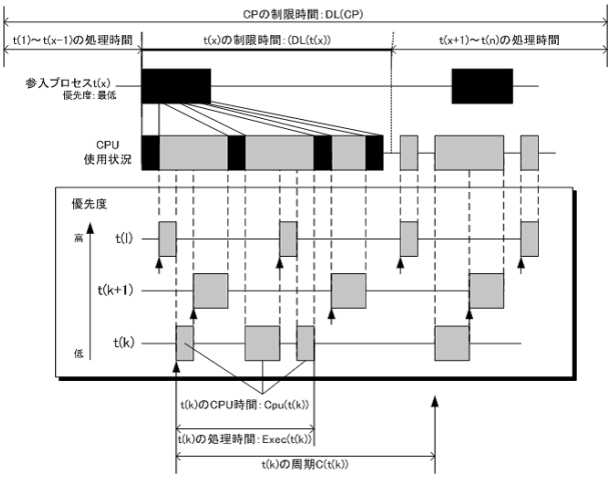

図 5 制限時間内での処理完了可能性判定方法

式 (1), (2) の判定を全ての計算機に対して行うことにより，プロセス t_x が制限時間内に動作可能な計算機を全て抽出する．

- (b) の判定方法：(a) の判定によって抽出した各計算機について，その計算機上で動作する，プロセス t_x よりも優先度の低いプロセスが全て制限時間内に処理完了可能かどうかを，式 (1), (2) を用いることにより判定する．優先度の低い全てのプロセスが制限時間内に処理完了可能であれば，その計算機はプロセス t_x に対して CPU リソースを割り当てることができる．
  なお，複数の計算機がリソース割当可能となった場合には，その中で「空き CPU リソース量」が最も小さい計算機を割当計算機として決定する．本研究における空き CPU リソースとは，最低優先度のプロセス t_1 が動作した場合の CPU の空き時間であり，計算機 A の空き CPU リソース：$Idle(A)$ は以下の式で求めることができる．

$$Idle(A) = DL(t_1) - \lceil \sum_{i=2}^{m} \frac{DL(t_1)}{C(t_i)} \rceil \times CPU(t_i) \quad (3)$$

手順 1-2：プロセス分割によるリソース割当計算機の選定：
次に，割当対象プロセスを小粒度に分割し，1 プロセスあたりの CPU 時間を小さくすることによりリソース割当を行う (**図 6**)．プロセス t_x を n 分割することにより，制限時間 DL は変わらずに CPU 時間は約 $1/n$ となる．したがって，

$$DL(t_x) - \lceil \sum_{i=l+1}^{m} \frac{DL(t_x)}{C(t_i)} \rceil \times Cpu(t_i) - \frac{Cpu(t_x)}{n} > \Delta$$

を満たす計算機が n 台以上あれば，プロセス分割によってリソース割当が可能となる．なお，割付可能計算機が

n 台以上あった場合には, 式 (3) を用いて, 空き CPU リソース量の少ない計算機を n 台選択すればよい.

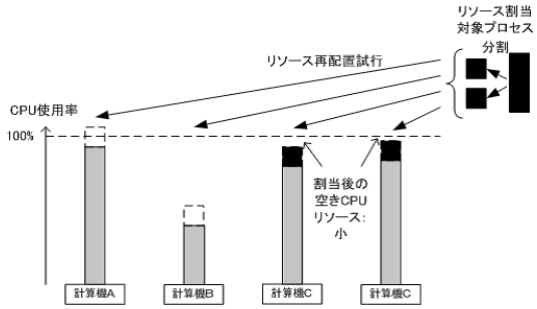

図 6 手順 1-2. の操作

手順 1-3：システム全体のリソース配置変更： 次に, システム全体のリソースを再配置することにより空きリソースを作成し, デッドラインミス発生の原因となったプロセスにリソースを割り当てる (**図 7**).

具体的には, ある計算機でリソース割当対象のプロセス t_x よりも優先度が高く, 実行時間の短いプロセス (t_y とする) を任意に選択する. そして, プロセス t_y を他の計算機のリソースを割り当てることにより計算機リソースの空きを作成し, プロセス t_x を制限時間内に処理が完了できるようにする. 次に, プロセス t_y のリソース割当計算機を同様の方法で決定する. これらの操作の結果, t_x, t_y がともに動作可能なリソース再配置パターンが存在した場合, それらの計算機がリソース割当先となる.

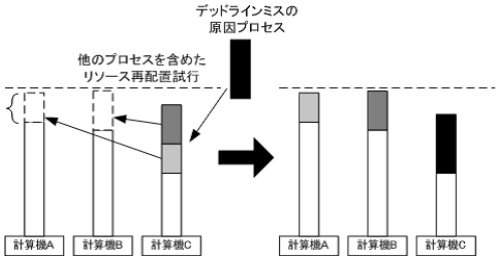

図 7 手順 1-3. の操作

手順 1-1 ～ 1-3 によって割当候補となった計算機群のうち, 最も空きリソースの少ない計算機の組み合わせをリソース割当計算機として決定する. なお, 本操作は全て行う必要はなく, デッドラインミスまでの残り時間に応じて行う.

手順 2：低重要度処理の停止による空きリソースの作成：

手順 1 でリソース割当計算機が決定しない場合には, 全ての処理を行えるリソースがシステム内にないと判断する. この場合には, 低負荷のクリティカルパスの処理を順次停止させてシステムに空きリソースを作成した上で, 式 (2), (3) を用いて適切なリソース割当計算機を決定する (**図 8**).

以上のように, デッドラインミス発生時刻を予想し, その時刻よりも前にリソース再割当を完了することにより, デッドラインミスを未然に防止することができる. また, 各計算機

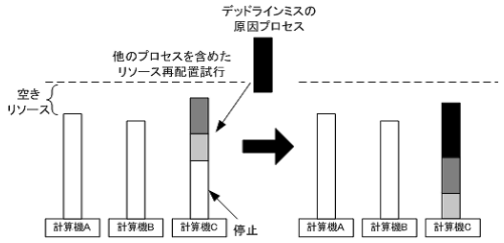

図 8 手順 2 の操作

の CPU リソースの空きが極力小さくなるようにリソース再配置を行うことによってシステムの空きリソースを 1 つの計算機に集めることができる. 従って, アルゴリズムの手順 1. でリソース割当計算機を検出できるケースが増加し, デッドラインミスまでの残り時間が短い場合においてもハードリアルタイム処理継続の可能性を向上させることができる.

3.2.3 待機プロセスによるデータのバックアップ

前述の通り, 本研究では「待機プロセス」を配置し, 待機プロセスと実際に処理を行っているプロセスとの間でデータを整合することにより, リソース再割当時における演算データの継続を実現する. 待機プロセスの管理, 処理中のプロセスと待機プロセスの間のデータ整合は以下のようにして行う.

待機プロセスの管理：

待機プロセスは任意の計算機で動作させることが可能であるが, 本研究では, リソースの浪費を極力小さくするため, リソース割当が想定される計算機に限定して配置することとしている. 具体的には, 各プロセスについて 3.2.2節に示した方法を用いることにより, デッドラインミスが発生した場合に割付可能な計算機を複数台選び出し, それらの計算機すべてに待機プロセスを配置する.

待機プロセスの起動も任意のタイミングで行うことができるが, 本研究では待機プロセスの起動が及ぼすオーバーヘッドを考慮し, デッドラインミスなどが発生して待機プロセスに処理を割り当てる度に, 割り当てた数だけ新規に起動することとし, 待機プロセスが少なくともシステム内に 1 つ以上存在するようにしている.

このように, リソース割当が想定される計算機に限定して待機プロセスを配置することにより, リソースの消費を極力防止し, さらに, リソース割当時にプロセスを起動する処理を省略できるためにリソース割り当てに要する時間が短縮され, デッドラインミス発生の危険性を小さくすることが可能になる.

演算プロセスと待機プロセス間のデータ整合：

演算プロセスと待機プロセスの演算データの整合は以下のようにして行う (**図 9**).

- 同一処理を行う演算プロセスとその待機プロセスで通信グループを作成し, グループ内で結果データをマルチキャストすることにより実現する.
- 上記に加え, 各通信グループ内にデータ保持プロセスを動作させる. データ保持プロセスは, 過去 N 周期の演算データを保持し, 新規起動した待機プロセスに対してデータを送信する役割を持つ.

本設計により, 待機プロセスが演算プロセスと同一のデータを保持できるため, 任意のタイミングでリソース割当変更が行われても同一のデータを用いて処理を継続することが可能である. また, 通信グループ内で過去のデータを保持することにより, 新規に起動した待機プロセスも他の待機プロセスと同じデータを保持することが可能になる.

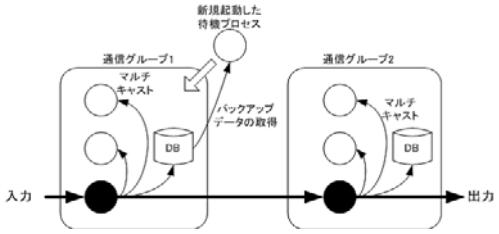

図 9 演算プロセスと待機プロセスのデータ整合

4. リソース管理ミドルウェア実装概要

2 章の設計内容と, 図 2 の構成をもとに作成した, 本研究におけるリソース管理ミドルウェアのコンポーネント構成を**図 10** に示す. 図中網掛け部分が今回の新規実装部分であり, 本ミドルウェアは, 従来の構成に加え, Replica Manager, DeadlineMiss Forecaster が追加されている.

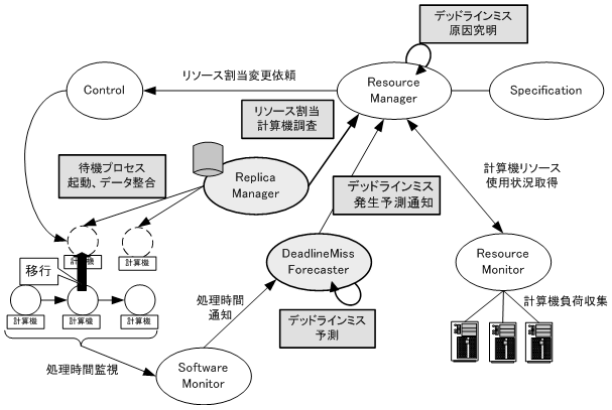

図 10 設計したリソース管理ミドルウェアのコンポーネント構成

本研究にて新規に追加したコンポーネント：DeadlineMiss Forecaster, Replica Manager の機能は以下の通り：

DeadlineMiss Forecaster: Deadline Forecaster は 3.2.1節に示す役割を持つ. 具体的には, Software Monitor が取得した各クリティカルパスおよびプロセスの処理時間を蓄積し, 時刻と処理時間の変動に関する近似式を求める. そして, 近似式を用いることによりデッドラインミス発生時刻を求め, Resource Manager にその時刻を通知する.

Replica Manager： Replica Manager は 3.2.3節に示した機能を持つ. 具体的には, Resource Manager に問い合わせることにより, 各プロセスについて, 将来リソース割当を行う可能性がある計算機を洗い出し, その計算機上でプロセスを起動する. また, 各プロセスが処理した過去 N 周期のデータを保持し, 新規に起動した待機プロセスに過去のデータを送信する.

本リソース管理ミドルウェアにおける, デッドラインミス検出からリソース再割付完了までの動作の一例を示す.

(1) DeadlineMiss Forecaster は, 全クリティカルパスの処理時間からデッドラインミス発生時刻を予想し, その時間が近づいたら Resource Manager にデッドラインミスの警告を Resource Manager に通知する.

(2) Resource Manager は, (1) の通知を受けると, プロセスの処理時間をもとにデッドラインミス発生の原因となったプロセスを検出する.

(3) Resource Manager は, 各計算機の空きメモリ量, CPU 空き時間を調査し, 3.2.2節に示した方法によってデッドラインミスを発生させることなく処理が可能な計算機を選択する.

(4) Resource Manager は, デッドラインミス発生の原因となったプロセスを, (3) で選択した計算機上に移行するよう Control に指示する.

(5) Control は, あらかじめ Replica Manager によって起動してあった待機プロセスに対して処理を割り当て, これまで処理していたプロセスを停止させる.

(6) Replica Manager は, Resource Manager に問い合わせることにより, (5) でリソース割付を行ったプロセスにリソース割付が可能な計算機の一覧を受け取り, その計算機上で新規に待機プロセスを起動する.

5. おわりに

本稿では, システム負荷の上限が予測できない分散リアルタイムシステムにおいて, リソース配置の動的な変更を行うことにより, アプリケーションのハードリアルタイムを実現するミドルウェアの設計を行った. 現在, 本設計内容をもとにミドルウェアのプロトタイプ実装を行っている.

今後, リソース管理ミドルウェアの処理オーバーヘッドなどについて調査を行うことにより, ハードリアルタイム処理の実現可能性について評価を行う予定である.

参 考 文 献

1) L. R. Welch et al., "Adaptive Resource Management for Scalable Dependable Real-Time Systems.", 4th IEEE Real-Time Technology and Applications Symposium , 1998.

2) Kevin Bryan, et al., "Integrated CORBA Scheduling and Resource Management for Distributed Real-Time Embedded Systems", 11th IEEE Real-Time and Embedded Technology and Applications Symposium, 2005.

3) B. Ravindran and P. Li,"DPR, LPR: Proactive Resource Allocation Algorithms for Asynchronous Real-Time Distributed Systems",IEEE Transactions on Computers, Volume 53, Number 2, pages 201-216, February 2004.