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

2008－SLDM－134／2008－EMB－8（32）
2008 / 3 / 28

RTミドルウェア用の優先度によるオブジェクト管理機構

千代 浩之† 武田 瑛† 上山 真生† 加藤 真平† 山崎 信行†

† 慶應義塾大学
〒 223-8522 神奈川県横浜市港北区日吉 3-14-1
E-mail: †{chishiro,takeda,ueyama,shinpei,yamasaki}@ny.ics.keio.ac.jp

あらまし 並列分散処理のオブジェクト指向ネットワークアプリケーションである分散制御ロボットには，ミドルウェアによる分散オブジェクト管理基盤の構築，開発の低コスト化，システムの拡張性，リアルタイム性の保証などが求められる．Robot Technology(RT) ミドルウェアは，RT コンポーネントを組み合わせることにより，多様なロボットシステムの統合を可能としたソフトウェアである．しかしながら，RT ミドルウェアにはミドルウェアレベルでリアルタイム性を保証する機構がない．そこで本研究では RT ミドルウェアを対象として，ミドルウェアレベルで時間予測性を向上させるために，優先度によるオブジェクト管理機構を提案する．RT コンポーネントはタスク情報を追加したメッセージを送信することで，オブジェクトを優先度順で実行可能となる．本研究の提案手法が従来手法と比較して，タスク成功率が向上することを示す．

キーワード RT ミドルウェア，CORBA，ARTLinux，リアルタイムシステム

Prioritized Object Management for RT-Middleware

Hiroyuki CHISHIRO†, Akira TAKEDA†, Masao UEYAMA†, Shinpei KATO†, and Nobuyuki YAMASAKI†

† Keio University
3-14-1, Hiyoshi, Kouhoku-ku, Yokohama, Kanagawa 223-8522 Japan
E-mail: †{chishiro,takeda,ueyama,shinpei,yamasaki}@ny.ics.keio.ac.jp

Abstract Distributed control robots, object-oriented network applications for parallel distributed processing, require distributed objent management infrastructure, low-cost development, system scalability, and real-time guarantee. RT-Middleware is software to integrate multiple robot systems by combining RT-Components. However, RT-Middleware does not have real-time guarantee in middleware layer. In this paper, we propose prioritized object management to improve time predictability for RT-Middleware in middleware layer. RT-Component send messages with task information, so objects can execute in order of priority. The evaluation shows that our method improves task success ratio than ever before.

Key words RT-Midddlware, CORBA, ARTLinux, real-time systems

1. はじめに

近年，ロボット技術の急速な進歩によりシステムが複雑化するため，集中制御では負荷が増大することが問題である．分散制御はこの問題に対する有効な解決方法であり，負荷の軽減による実時間性の保証，耐故障性や拡張性を向上することが可能である．

しかしながら，実際に分散制御ロボットを開発し，複数のプラットフォームに移植するとなると，本質的問題と副次的問題が発生する．本質的問題とは，予測可能で信頼性，効率性の高いシステムを実現するために効果的な並列処理の利用，システムの可用性と柔軟性を最大にするために，サービスを実行環境ごとに構成，配置しなければならないことである．副次的問題とは，型安全で可搬性のある拡張性の高いネイティブ OS API の欠如，手続き指向設計のようにアルゴリズムによる分割が行われているため，移植性，拡張性，再利用性が低いことである．分散制御ロボットの開発者はこれらの問題を理解し，オブジェクト指向設計のようにデータによる分割を行い，効率的に対処

しなければならない．このような問題は，ミドルウェアを用いることで解決することができる．ミドルウェアとは，OS とアプリケーションの中間に位置するソフトウェアである．ミドルウェアを用いて OS に非依存なプログラムを書くことにより，移植性の向上，ソースコードの再利用が容易になる．

Robot Technology(RT) ミドルウェア [1] は，RT コンポーネントを組み合わせることで，多様なネットワークロボットシステムの統合を可能にし，ネットワーク分散コンポーネント化技術による共通プラットフォームを確立するためのソフトウェアである．分散制御ロボットには，時間粒度の細かい協調動作を実現するため，ミドルウェアレベルでリアルタイム性を保証する必要がある．しかしながら，従来の RT ミドルウェアでは，モジュールロボットの制御で必要となるミドルウェアレベルでリアルタイム性を保証する機構がない．そこで本研究では RT ミドルウェアを対象とし，ミドルウェアレベルでリアルタイム性を保証するために，優先度によるオブジェクト管理機構を提案する．その有効性を，デッドラインまでにタスクの実行が完了した割合である，タスク成功率の観点から評価する．

本論文の構成は以下の通りである．第 2 章では，RT ミドルウェアと Common Object Request Broker Architecture(CORBA) [2] について述べる．第 3 章では，本研究の提案した優先度によるオブジェクト管理機構について述べる．第 4 章で本研究の提案手法の評価及び考察について述べる．第 5 章で結論と今後の課題について述べる．

2. RT ミドルウェア

RT ミドルウェアとアプリケーションである RT コンポーネント (RTC) を含めたソフトウェアアーキテクチャを図 1 に示す．

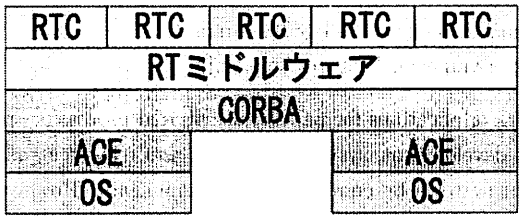

図 1　ソフトウェアアーキテクチャ

RT ミドルウェアとは，RT 機能要素のソフトウェアモジュールを複数組み合わせて RT システムを構築するためのソフトウェアプラットフォームである．RT 機能要素をソフトウェアモジュール化したものを RT コンポーネントという．ロボットのモジュールを RT コンポーネント単位で実装することにより，他のミドルウェアと比較して，再利用性，拡張性，移植性の高い実装が可能となる．

RT ミドルウェアはホスト基盤ミドルウェアである ADAPTIVE Communication Environment(ACE) [3] と，CORBA を実装した分散ミドルウェアである omniORB を用いて実装されている．CORBA とは，Object Management Group(OMG) [4] によって標準化されている分散オブジェクト技術の仕様である．CORBA は環境や言語に依存しないオープンな仕様であり、複数の環境や言語の混在した分散アプリケーションを構築することが可能である．CORBA オブジェクトは実装と定義 (インターフェース) が分離している．インターフェースは Interface Description Language(IDL) によって記述され，オブジェクトの実装は完全に外部から隠蔽されている．したがって，オブジェクトがどのようなプログラム言語で実装されていたとしても，その言語と IDL のマッピングが定義されている限り，一様に CORBA オブジェクトとして扱うことができるので，言語非依存の実装が可能となる．

RT ミドルウェアは ACE，omniORB を利用することで RT 機能用のインターフェース部分のみを実装すればよい．したがって，RT ミドルウェアは既存のミドルウェアを効果的に利用することで，RT ミドルウェア独自の実装は小規模で様々な機構を利用可能となる．ソフトウェアの開発コストを削減し，より高度な実行環境を提供するために，将来はミドルウェアを利用したミドルウェアが主流になると考えられる．

RT コンポーネント間でメッセージを送受信するためのポートとして，データポートとサービスポートがある．データポートとは，データ指向通信をサポートする Publisher/Subscriber モデルに基づき RT コンポーネント間のデータの送受信を抽象化する．サービスポートとは，IDL によってユーザ定義の任意のサービスインターフェースを持たせることが可能である．ロボットの開発者は煩雑なネットワークプログラミングを行わずに，メッセージの送受信，オブジェクトの位置や型を意識せずに実装が可能となる．したがって，RT ミドルウェアを用いてロボットを開発することは，低コスト化や汎用化のメリットがあると考えられる．

従来の RT ミドルウェアでは，ミドルウェアレベルでリアルタイム性を保証する機構がない．例えば，RT コンポーネント間の協調動作をする場合，デッドラインを保証できない場合がある．本研究では，この問題を解決するために，ミドルウェアレベルで優先度によるオブジェクト管理を行う．

3. 優先度によるオブジェクト管理機構

本研究の提案した優先度によるオブジェクト管理機構について述べる．本研究では，ロボットのモジュール毎に単一の CPU から構成されるシステムを想定する．

本研究の周期タスクは 2 種類あり，それぞれをローカルタスク，リクエストタスクと呼ぶ．ローカルタスクとは，リクエストタスクを送信，またはロボットのモジュールの位置制御などのように自分の RT コンポーネント内で完結するタスクである．リクエストタスクは，他の RT コンポーネントに処理を依頼するタスクである．

3.1 リアルタイムスレッド

従来の Linux のスレッドではタスクの実行中に，割り込み等により実行を妨害されてしまい，時間予測性が低下してしまう問題がある．この問題を解決するために，周期タスクを実行するために必要なスレッドは，ARTLinux [5] によりリアルタイム処理を実行可能なスレッドにする．ARTLinux とは，

スレッド	優先度
サーバ	高
ローカル	中
ワーカー	低

表 1　リアルタイムスレッドの優先度

ハードリアルタイム処理機能を拡張した Linux カーネルである．ARTLinux の特長として，プリエンプティブマルチタスク機能，固定優先度によるスケジューリング機能を持つ．また，多段階の優先度継承機能，デバイスドライバのソース，アプリケーションのバイナリと互換性がある．リアルタイムタスクをユーザプロセスで実行するため，RTLinux [6]，RTAI [7] と違い安全にリアルタイム処理を実行可能である．

本研究の実装で ARTLinux によりリアルタイム実行するスレッドは 3 種類ある．それぞれをサーバースレッド，ローカルスレッド，ワーカースレッドと呼ぶ．リアルタイムスレッドの優先度を表 1，それらの特長を以下に示す．

- サーバースレッド: 一定時間ごとにリクエストタスクが届いているかどうかを監視する．
- ローカルスレッド: 周期毎にローカルタスクを実行する．
- ワーカースレッド: サーバースレッドにより起動されることにより，リクエストタスクを受信し，実行する．リクエストタスクの実行終了時に新しいリクエストタスクが届いている場合は，サーバースレッドを介さずに，続けてリクエストタスクを実行する．

これらのリアルタイムスレッドの中で，サーバースレッドを最高優先度にしなければ，リクエストタスクの受信処理が他のリアルタイムスレッドに妨害されてしまい，時間予測性が低下すると考えられる．また，ローカルスレッドをワーカースレッドより優先度を高くしなければ，ローカルスレッドのリクエストタスクの送信処理がワーカースレッドによるリクエストタスクの実行処理に妨害されてしまい，デッドラインミスが多くなると考えられる．したがって，リアルタイムスレッドの優先度を高い順にサーバー，ローカル，ワーカースレッドに設定する．

3.2　タスク管理モデル

RT ミドルウェアのタスク管理を図 2 に示す．

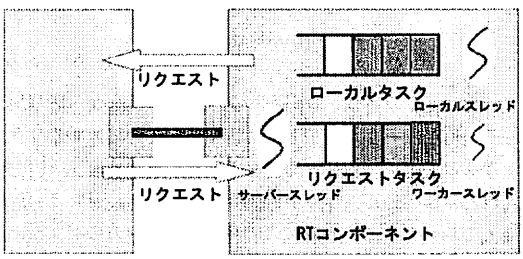

図 2　RT ミドルウェアのタスク管理

RT ミドルウェアでは，ローカルタスクとリクエストタスクを管理する．ローカルタスク内，リクエストタスク内ではノンプリエンプティブである．つまり，あるローカルタスクの実行中に他のローカルタスクがプリエンプティブして実行することはできない．このモデルはプリエンプティブ方式と比較して，優先度の低いタスクが優先度の高いタスクの実行を妨害することによる優先度逆転問題が発生する欠点がある．しかしながら，コンテキストスイッチによるオーバーヘッドを削減し，スループットを向上させる利点があるので，本研究ではノンプリエンプティブ方式を採用する．

ローカルタスクは RT ミドルウェアで管理する．従来の RT ミドルウェアでは，1 つの RT コンポーネントには 1 つのローカルタスクしか実行できないので，複数のローカルタスクを実行する場合は，ローカルタスクの数だけ RT コンポーネントを生成し，ロボットの協調動作を行う場合は該当する RT コンポーネントを全て接続しなければならないので，システム稼動時のコストが大きく，またアプリケーションレベルのプログラミングが複雑になるので，分散制御ロボットで利用することは不適切だと考えられる．この問題を解決するために，本研究では 1 つの RT コンポーネントに複数のローカルタスクの実行を可能にするために，従来の RT ミドルウェアを拡張した．

リクエストタスクは RT ミドルウェアではなく，omniORB で管理する．従来の omniORB はリクエストタスクは FIFO で実行する．この実装では，優先度の高いリクエストタスクが優先度の低いリクエストタスクの実行を妨害してしまう．したがって，本研究ではリクエストタスクを優先度により実行順序を制御する．omniORB にはワーカースレッドを管理する機構として，スレッドプールとスレッドパーコネクションがある．スレッドプールとは，リクエストタスクを並行に実行できるワーカースレッドの集合を確保するための並列モデルである．ワーカースレッドを予め生成することによりシステム稼動時のワーカースレッドの生成コストを削減することが可能である．しかしながら，システム初期化時のワーカースレッドの生成コストがスレッドパーコネクションと比較して大きいという欠点がある．スレッドパーコネクションとは，ネットワークの接続ごとに別々のワーカースレッドを関連付ける並列モデルである．スレッドパーコネクションは，接続時にワーカースレッドを生成するので，スレッドプール方式と比較してシステム稼動時のオーバーヘッドが大きいという欠点があるが，システム初期化時のコストが小さい．本研究では，システム稼動時を想定して実装するので，スレッドプール方式を採用する．また，それぞれのリアルタイムスレッドのスレッド数を 1 つに制限する．つまり，サーバースレッドはリクエストタスクを 1 つのワーカースレッドに割り当てる．リクエストタスクを実行するワーカースレッドをシステム起動時に予め生成することにより，予測可能性を向上させ，実装を簡単にする．

3.3　タスク情報の交換

RT コンポーネントが持つタスク情報は他の RT コンポーネントとの接続時に交換する．リクエストタスクの送信毎にタスク情報を送信する場合と比較して，タスクの識別番号のみでリクエストタスクを実行することが可能となるので，パケットのサイズが小さくなり，システム稼動時のオーバーヘッドが小さくすることが可能となる．タスク情報を交換するために必要な実

装である，rtTask クラスを図 3，タスク群の情報を一括管理する rtTaskPool クラスを図 4，接続時に交換する rtTaskPacket 構造体を図 5 に示す．紙面の都合上，重要でないメンバ変数，メンバ関数，コンストラクタ，デストラクタ等は省略する．

```
class rtTask {
public:
  rtTask(CORBA::ULong period,
         CORBA::ULong exec_time,
         CORBA::ULong rq_exec_time,
         CORBA::ULong cb_exec_time,
         char comp_name[32]);
  void setPeriod(CORBA::ULong period)
  void setExecTime(CORBA::ULong exec_time);
  void setRqExecTime(CORBA::ULong rq_exec_time);
  void setCbExecTime(CORBA::ULong cb_exec_time);
  void setCompName(char comp_name[32]);
private:
  CORBA::ULong pd_period;
  CORBA::ULong pd_exec_time;
  CORBA::ULong pd_rq_exec_time;
  CORBA::ULong pd_cb_exec_time;
  CORBA::ULong pd_comp_name[32];
};
```

図 3　rtTask クラス

```
class rtTaskPool {
public:
  void add(CORBA::rtTask task);
 private:
  std::vector<CORBA::rtTask> pd_task;
};
```

図 4　rtTaskPool クラス

```
struct rtTaskPacket {
  CORBA::ULong period[64];
  CORBA::ULong exec_time[64];
  CORBA::ULong rq_exec_time[64];
  CORBA::ULong cb_exec_time[64];
  char comp_name[32];
};
```

図 5　rtTaskPacket 構造体

rtTask クラスは後述する rtTaskPool クラスにタスクを登録するためのクラスである．メンバ変数 pd_period はタスクの周期，pd_exec_time はローカルタスクの実行時間，rq_exec_time はリクエストタスクの実行時間，cb_exec_time はリクエストタスクの終了時に実行するコールバックタスクの実行時間，comp_name は RT コンポーネントの名前を表す．本研究の実装では，rtTask クラスのメンバ変数はコンストラクタか，または setter メンバ関数を用いて登録する．

rtTaskPool クラスは先述した rtTask クラスを一括管理するためのクラスである．メンバ変数 pd_task は rtTask クラスをクラステンプレートとする vector である．add メンバ関数は rtTask クラス変数 task をメンバ変数 pd_task に追加する．

rtTaskPacket 構造体は，RT コンポーネントの接続時に交換するタスク情報を持つ構造体である．rtTask クラスと同様のメンバ変数を持つ．本研究の実装ではタスク数は最大 64 までとする．

3.4　ローカルタスク

ローカルタスクの概要とリクエストタスクに追加して送信する rtTaskInfo 構造体をそれぞれ図 6，図 7 に示す．

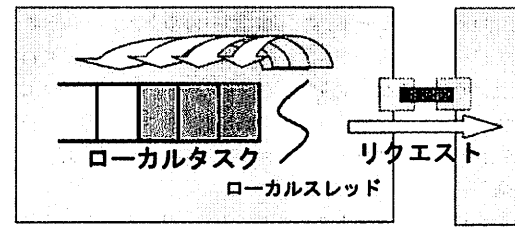

図 6　ローカルタスクの概要

```
struct rtTaskInfo {
  CORBA::ULong task_id;
  CORBA::ULong msz;
  CORBA::ULong exec_time;
  CORBA::ULong period;
  char comp_name[32];
};
```

図 7　rtTaskInfo 構造体

ローカルスレッドは，ローカルタスクをキューから取り出さずに優先度キューから選択して実行するので，後述するリクエストタスクと違い，ローカルタスクは実行後でもキュー内に保持される．ローカルスレッドの周期は全ローカルタスクの周期の最大公約数とする．本研究の実装では，ローカルタスクの周期の短いタスクに高い優先度を割り当てる．同じ優先度の場合は先に到着したローカルタスクを先に実行する．

ローカルタスクは，CORBA オブジェクトに関する情報である General Inter-ORB Protocol(GIOP) メッセージに図 7 のタスク情報を載せてリクエストタスクを送信する．rtTaskInfo 構造体は，メンバ変数 task_id はタスクの種類，メンバ変数 msz

は rtTaskInfo 構造体と GIOP メッセージの合計サイズ，メンバ変数 exec_time は周期タスクの実行時間，メンバ変数 period は周期タスクの周期，メンバ変数 comp_name は RT コンポーネントの名前を表す．メンバ変数 comp_name は，リクエストタスクの送信した RT コンポーネントをを判別するために必要である．受信バッファは 8K バイトの固定長なので，実際に届いているタスク情報付き GIOP メッセージが 8K バイトより多い時は，最後のタスク情報付き GIOP メッセージが不完全な状態で受信バッファに格納されてしまう．この問題を解決するために，メンバ変数 msz は最後に受信したタスク情報付き GIOP メッセージが不完全なことを検知し，完全な GIOP メッセージにするために必要なパケットサイズを算出するために必要である．

3.5 リクエストタスク

リクエストタスクの概要を図 8 に示す．

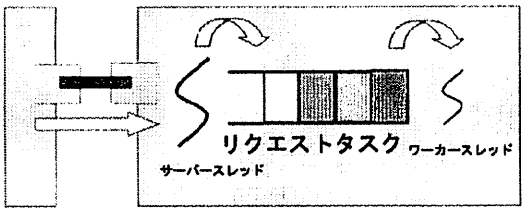

図 8 リクエストタスクの概要

リクエストタスクは，サーバスレッドにより優先度キューに挿入され，ワーカースレッドによりキューから取り出される．本研究の実装では，周期が短いタスクに高い優先度を割り当てる．優先度が同じ場合は，先に到着したリクエストタスクを先に実行する．リクエストタスクを優先度キューに挿入する時に優先度順にソートし，キューの先頭にあるタスクをキューから取り出す．リクエストタスクの実行手順をアルゴリズム 1 に示す．

アルゴリズム 1 リクエストタスクの実行手順

```
1: if buffer receives GIOP messages with request task information then
2:   repeat
3:     if tail GIOP message with task information is broken then
4:       recv necessary size of packet for broken GIOP message with task information
5:       add necessary size of packet to broken GIOP message
6:     end if
7:     split GIOP messages with request task information into GIOP message and request task information
8:     enqueue GIOP message into priority queue by request task information
9:   until buffer is all demultiplexed
10: end if
11: dequeue request task from priority queue
12: execute request task
```

アルゴリズム 1 は，バッファにタスク情報を付加した GIOP メッセージを受信した場合 (1 行目)，タスク情報付き GIOP メッセージが不完全ならば (3 行目)，完全なタスク情報付き GIOP メッセージにするためのサイズのみパケットを受信し (4 行目)，不完全なタスク情報付き GIOP メッセージにパケットを追加して，完全なタスク情報付き GIOP メッセージにする (5 行目)．タスク情報を付加した GIOP メッセージ群から GIOP メッセージを分離し (7 行目)，GIOP メッセージをタスク情報により優先度キューに挿入する (8 行目)．バッファにある全ての GIOP メッセージが逆多重化されるまで繰り返す (9 行目)．リクエストタスクを優先度付きキューから取り出し (11 行目)，リクエストタスクを実行する (12 行目)．ワーカースレッドはリクエストタスクをキューから取り出して実行するので，先述したローカルタスクと違い，実行後に削除される．

4. 評　　価

周期タスクの周期や CPU 使用率を変えた場合のリクエストタスク成功率を評価する．評価に用いる 2 台のマシンは CPU はそれぞれ Pentium 4 2.53GHz，Pentium 4 2.26GHz，メモリはそれぞれ 512MB，768MB である．OS は Ubuntu7.10，カーネルは ART-2.6.22，コンパイラは g++-4.1.3，RT ミドルウェアは OpenRTM-aist-0.4.1，omniORB-4.0.7，ACE-5.6 を用いる．

4.1 評価方法

マシンをロボットのモジュールと想定して，マシン 1 台につき RT コンポーネントは 1 つとする．2 つの RT コンポーネントは共に 1 つのサービスポートで接続する．本研究では RT コンポーネント同士の接続後の状態で評価するので，接続時にかかるオーバーヘッドは考慮しない．評価方法の概要を図 9 に示す．

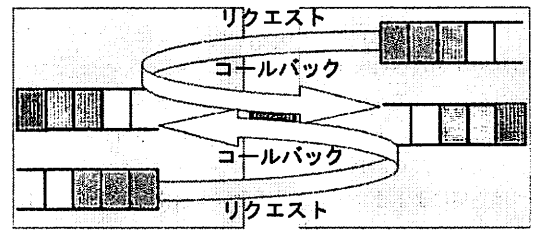

図 9 評価方法の概要

2 つの RT コンポーネント内でローカルタスク，RT コンポーネント間でリクエストタスクを実行する．評価指標としてのリクエストタスク成功率を以下の式に示す．

$$\text{リクエストタスク成功率} = \frac{\text{周期内に終了したリクエストタスク数}}{\text{総リクエストタスク数}}$$

周期内にリクエストタスクの実行が完了したか判定するために，リクエストタスクの実行の最後に，コールバックのためのリクエストタスクであるコールバックタスクを実行し，コールバックタスクの終了までの時間を評価する．サービスポートのインターフェースを図 10 に示す．

変数 task_id により接続時に交換したタスクの周期と実行時間を取得する．変数 seq_num はタスクのシーケンス番号を表す．

```
interface RTRequest {
  oneway void request(in unsigned long task_id,
                      in unsigned long seq_num);
};
interface RTCallback {
  oneway void callback(in unsigned long task_id,
                       in unsigned long seq_num);
};
```

図 10 サービスポートのインターフェース

リクエストタスクはノンブロッキングで送信するため，request オペレーション，callback オペレーションには oneway 属性を付加する．RTRequest インターフェースは相手の RT コンポーネント，RTCallback インターフェースは自分の RT コンポーネントで実装する．

周期タスク数はローカルタスク数は 3 に固定し，リクエストタスク数を RT として RT={3, 5} の場合を評価する．リクエストタスクのラウンドトリップタイムは約 10ms である．タスクの周期が 100ms 未満だと，ラウンドトリップタイムの影響が大きくなり，不適切な結果になると考えられる．したがって，本研究ではタスクの周期は 100ms 以上に設定する．

タスクの周期は式 $T \in \{100, 150, 200, ..., 1000\}$，実行時間は式 $C \in \{10, 15, 20, ..., 300\}$ から一様分布により選択する．実行時間は，全タスクの周期の最小公倍数であるハイパーピリオドを経過した時点でタスクの実行を終了する．全タスクの CPU 使用率を 0.3〜1.0 まで 0.05 刻みに変化させた場合のリクエストタスク成功率を評価する．

4.2 評価結果

RT = {3, 5} の場合の CPU 使用率に応じたリクエストタスク成功率をそれぞれ図 11 と図 12 に示す．

全体のリクエストタスク成功率は，提案手法は従来手法と比較して等しいか，または高い．したがって，本研究の提案手法により実装した GIOP メッセージにタスク情報を付加することや，優先度によるオブジェクト管理によるオーバーヘッドは小さく，無視できるレベルであると言える．

従来手法と提案手法を比較して，RT = {3,5} の場合は共に CPU 使用率が 0.8 未満ならばリクエストタスク成功率はほとんど変化がない．しかしながら，0.8 を超えると従来手法より提案手法の方がリクエストタスク成功率が高く，RT = 3 の場合は最大 8%，RT = 5 の場合は最大 15%向上している．これは，優先度キューによりリクエストタスクの実行順序が到着順ではなく，優先度順に実行されるからであると考えられる．RT = 5 は RT = 3 と比較して同じ CPU 使用率でリクエストタスク成功率が低いのは，リクエストタスクの送受信処理や，リアルタイムスレッドのコンテキストスイッチによるオーバーヘッドが大きいからだと考えられる．したがって，本研究の実装はリクエストタスク数が多いほど有効であると考えられる．

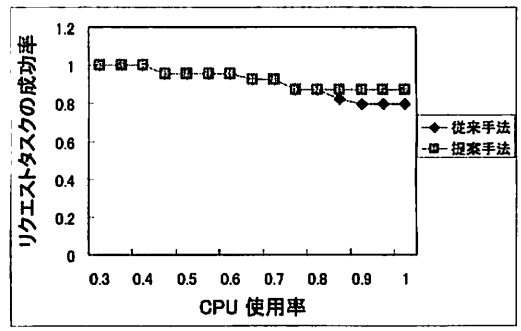

図 11 CPU 使用率に応じたリクエストタスク成功率 (RT = 3)

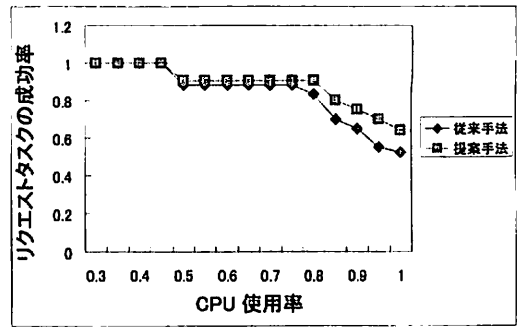

図 12 CPU 使用率に応じたリクエストタスク成功率 (RT = 5)

5. 結　論

本研究では，RT ミドルウェア用の優先度によるオブジェクト管理機構を提案した．オブジェクトにタスク情報を持たせることにより，オブジェクトの実行順序を決定することを可能とした．周期の短いタスクに高い優先度を割り当てることにより，周期タスクのタスク成功率が最大 15%向上した．また，従来の RT ミドルウェアと互換性を保つように設計及び実装したことにより，既存の RT コンポーネントをほとんど変更することなく，ミドルウェアレベルでリアルタイム化することが可能である．本研究ではタスクは周期タスクのみと仮定したが，非周期タスクも考慮する必要がある．

謝辞 本研究は，産業技術総合開発機構 NEDO の支援による．

文　献

[1] N. Ando, T. Kitagaki, K. Kitagaki, T. Kotoku and W.-K. Yoon: "RT-Middleware: Distributed Component Middleware for RT (Robot Technology)", In Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems, 11, pp. 3933-3938 (2005).
[2] O. M. Group: "Common Object Request Broker Architecture: Core Specification", formal/04-03-12 edition (2004).
[3] D. C. Schmidt: "The ADAPTIVE Communication Environment", http://www.cs.wustl.edu/~schmidt/ACE.html.
[4] O. M. Group: http://www.omg.org/.
[5] M. Eye: "ARTLinux", http://www.movingeye.co.jp/.
[6] "RTLinux", http://www.rtlinux.org/.
[7] "RTAI", http://www.rtai.org/.