無線タグネットワークシステムの社会インフラ化に向けたシステム仮想化技術の研究

代表研究者　栗　林　伸　一　成蹊大学理工学部教授
共同研究者　甲　斐　宗　徳　成蹊大学理工学部教授

1 はじめに

ユビキタスネットワーク社会の進展に伴い、物やオブジェクトを識別するための無線タグ利用が拡大し、それを用いた多種多様なサービスはインターネットのように国民生活になくてはならないものとなってくると考えられる[1]-[4]。そのためには、無線タグネットワークシステムを電気、ガス、水道と同様に、社会インフラとして提供し全ての企業、官庁、国民が安く、安全にかつ安定的に利用できる環境を整える必要がある。

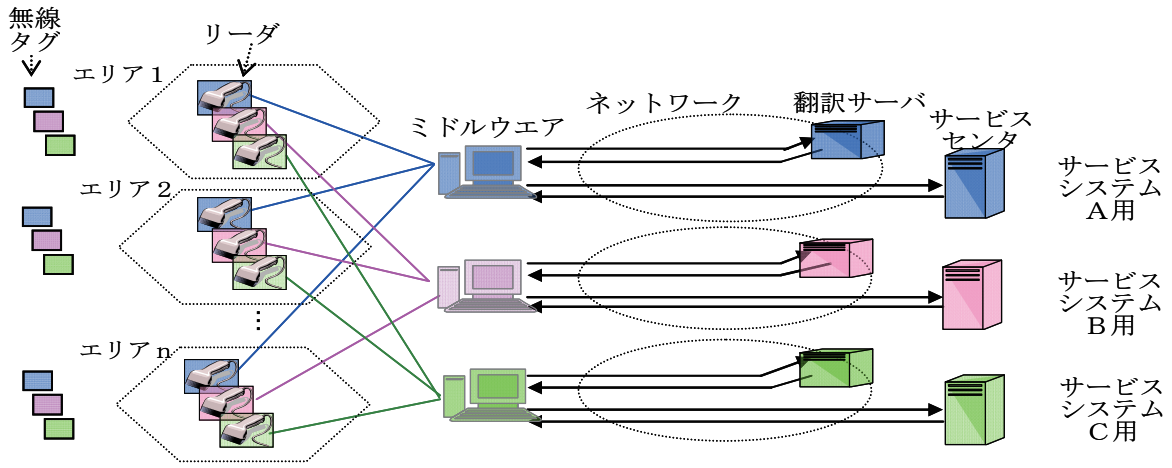
無線タグネットワークの仕様についてはＥＰＣグローバル[5]やユビキタスＩＤセンタ[6]などの標準化団体によって共通化が図られている。また、複数の異なる周波数やインタフェースを１台で収容可能とするリーダ装置共通化も進められている[7]。社会インフラとして安価なサービスを提供するためには、無線タグネットワークシステムの共用化が必須と考えられる[8]。また、社会インフラとして提供するためには、障害やふくそうなどによるサービス中断を極力無くす必要があり、プロセスマイグレーション技術（処理プロセスをネットワーク内の他コンピュータに簡単かつ確実に移動させる技術）が必須となる。

本研究は，ＥＰＣグローバルネットワークアーキテクチャを前提とし、従来個別に設計・構築されている無線タグネットワークシステムを共用し、かつそれを社会インフラとして提供するためのシステム共用化技術とプロセスマイグレーション技術を確立し、ユビキタスネットワーク社会の早期実現を図ることを目的とする。

2 無線タグネットワークシステム共用化技術

2-1 共用化の基本的考え方

本研究ではＥＰＣグローバルネットワークアーキテクチャ[5]を前提とする。図１に示すように、従来は会社毎、システム毎に個別に無線タグネットワークを構築することが多かった。無線タグネットワークを社会インフラとして提供するためには、システムを共用化することが必須となる。



＊同じエリアに、システム毎のリーダを複数重複して設置

図1. 現状の無線タグネットワークシステム

その共用化方法としては、装置を使用する時間帯をシステム毎に分割する方法（案ａ）と、装置内に複数の仮想化装置を構築しそれを各システムで使用する方法（案ｂ）が考えられる。実際の利用を考えると、システム毎に使用する時間帯を固定するのは現実的ではなく、案ｂの仮想化を前提とすることが望ましい。仮想化は装置の共用化によるコストダウンだけではなく、特にリーダなどの設置スペースや運用管理稼動の大幅な削減も期待できる。

ネットワークならびにサーバに関しては既に多くの仮想化技術が提案されており[9]、[10]、それらを無線タグネットワークシステムの各構成装置やネットワークに適用することが可能と考えられる。しかし、無線タグは１つの物に１つのタグを付けるのが基本であること、無線タグの制約された機能やメモリ量から考えると共用するのは困難であること、から、無線タグについては仮想化せずにシステム毎に準備するものとする。

これらを前提とした仮想無線タグネットワークシステムの構成イメージを図２に示す。システムを構成するリーダ、ミドルウエア、翻訳サーバ（タグＩＤからサービスセンタのアドレスを翻訳）、ネットワーク、サービスセンタ、などの装置は個々に仮想化され、それぞれの仮想化された装置要素を組み合わせることにより１つの仮想無線タグネットワークシステムを構築する。また、仮想化要素の生成ならびに管理はシステム管理サーバで一括して実施することを前提とする。例えば、システムＢ向け仮想無線タグネットワークシステムを構築する場合、システム管理サーバはシステムＢ向けシステム用件を精査し、その後図２に示すように必要な場所に必要な性能や容量を持つ仮想化要素をサービスセンタ、翻訳サーバ、ミドルウエア、リーダ内に生成する（関連制御データの設定も含む）。システム管理サーバは無線タグネットワークシステム内の全装置と全ネットワークの資源割り当て管理、運用管理を行う。なお、仮想リーダを生成するリーダは１つではなく、カバーすべきエリアの全でのリーダ内に仮想リーダを生成する。複数の仮想リーダと仮想ミドルウエアの連携情報は仮想ミドルウエアに設定される。ネットワークは専用のＶＰＮ(仮想専用網)を構築し、仮想化要素とＶＰＮの連携情報を各仮想化要素に設定する。逆に、特定システム向け仮想無線タグネットワークシステムを廃止する場合には、上記とは逆に関連する仮想化要素、ＶＰＮの解放を指示する。

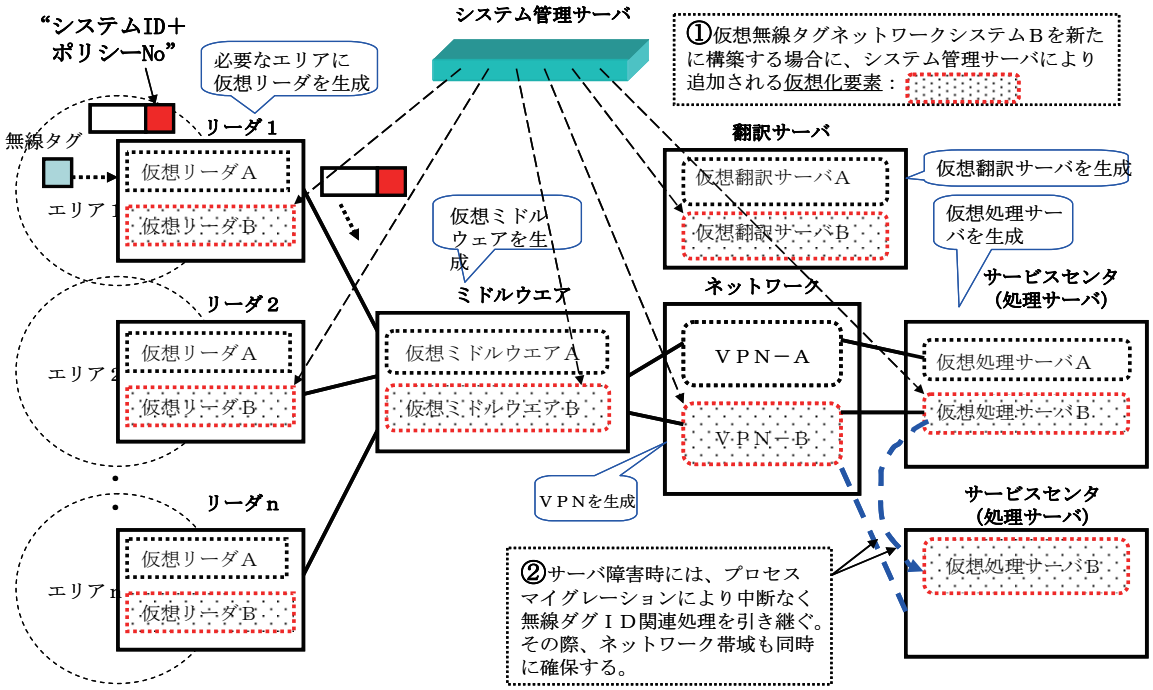


図２．社会インフラ化に向けた仮想無線タグネットワークシステム概要

２−２無線タグと仮想リーダの連携法

仮想無線タグネットワークシステムを実現するためには、仮想化しない無線タグと仮想化された仮想リーダとの連携が必要である（図３）。その連携法として以下の２案が考えられる（図４）。

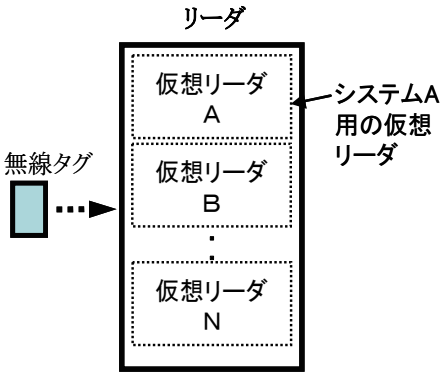


図3. 無線タグと仮想リーダの連携方式

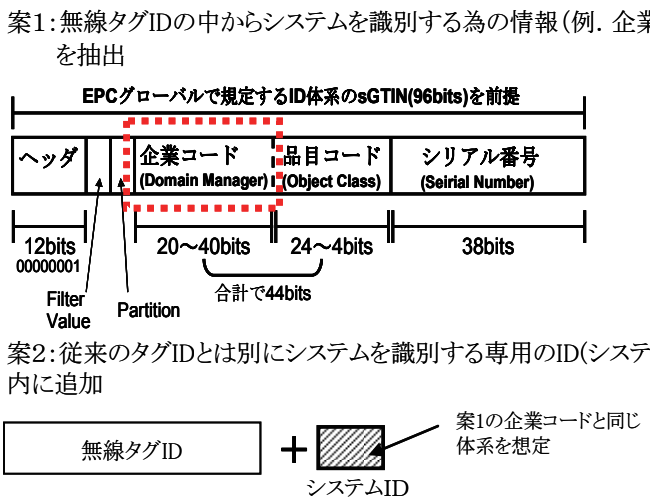


図4. 無線タグと仮想リーダの対応付け

案１：無線タグ内に格納されているタグＩＤ、物理リーダ内の仮想リーダ識別子など、既に存在する識別子からシステムやサービスを特定化する情報を抽出する。

案２：既に存在する識別子とは別に、システムやサービスを示す新たな識別子「**システムＩＤ**」を導入する。その識別子により対応関係を維持し、また必要な処理を実施する。なお、システムＩＤの大きさはタグＩＤコード体系などで定義される企業コード程度を想定する。

以下の理由から、案２を採用する。

１）既存の識別子にも様々なタイプがあり、それらを全て意識して対応するのは困難である。

２）案１は１つのサービスを複数の会社で提供する場合に対応できない。なお、複数企業を１つのグループにまとめたものを企業コードとして定義して使用することも可能ではあるが、その組み合わせが多く限界がある。

案２にもとづく処理イメージを図２を使って説明する。まず、システムＩＤをタグＩＤと一緒に無線タグ内に格納しておく。これを読みとったリーダは、該当する仮想リーダ機能が存在することを確認し、読み取ったタグＩＤとシステムＩＤをミドルウエアへ転送する。このシステムＩＤは、仮想ミドルウエア-→仮想ネットワーク-→仮想翻訳サーバ-→仮想処理サーバへと引き継ぐことにより、仮想化要素がどの物理装置内に設定されていても連携を取ることができることが判明した。なお、このシステムＩＤは仮想ネットワーク（ＶＰＮ）のＶＰＮ－ＩＤやＶＬＡＮ－ＩＤに類似したものであるが、システム全体としての連携を図るためのものであるため、それらとは別に転送する。

２-３社会インフラ化のための利便性向上策

現状方式では、時間帯によっては読み込んでも意味がない場合でもサービスセンタにタグＩＤが送られ、サービスセンタは処理することになる。これを、仮想リーダなどでカットすることによりサービスセンタでの不要な処理をなくすことが考えられる（図５）。従来でも一定数以上読み込むまでミドルウエアからサービスセンタにタグＩＤを転送しない仕組みは存在するが、時間帯や場所によって無効なタグＩＤ転送を仮想リーダで防ぐのが本研究で提案する方式のポイントである。

文献[1]～[4]をもとに、導入済またはデモ実施された複数の無線タグネットワークシステムを調査・分析した。その結果、無線タグのタグIDを読み取った後の動作としては、1)無条件にミドルウェアに送信、2)地域を指定して特別なアクションを要求、3)時間を指定して特別なアクションを要求、4)地域と時間の組み合わせを指定して特別なアクションを要求、に大きく分類できる。これを踏まえ、以下の4つの条件を考慮する。

＜考慮すべき条件＞

・特定のエリア（場所）にいる場合にミドルウエアに送信する。

・特定の時間帯に仮想リーダが読み込みとった時にミドルウエアに送信する。

・ネットワークがふくそう状態であっても指定したパケットは優先してミドルウエアに送信する。

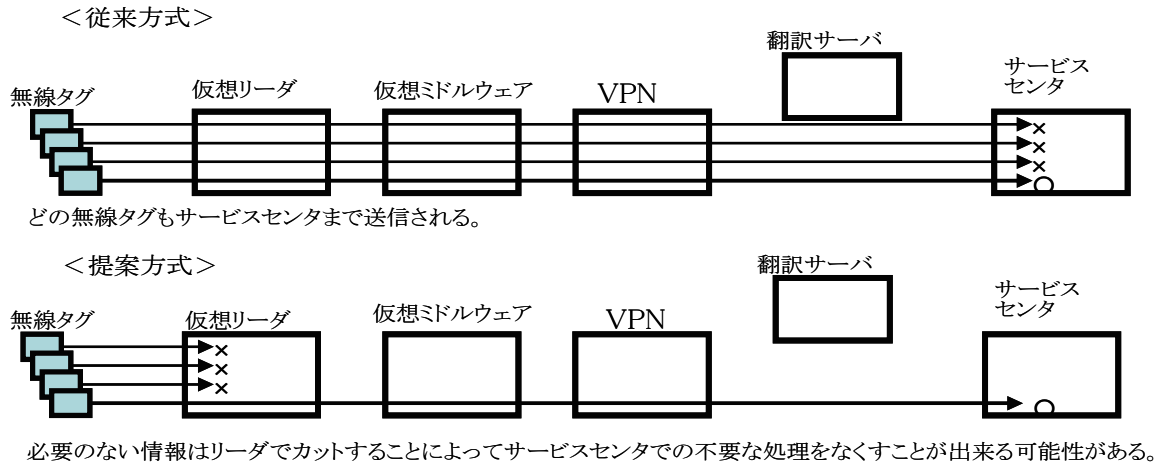


図5. 社会インフラ化のための利便性の向上策

・どの暗号化方式を採用しているか識別する。

上記を実現するための方式案を以下に示す。

案A： 仮想リーダがタグＩＤ毎の処理の違いを管理する。

案B： １つの無線タグに複数のタグＩＤを割り当てる。

案C： 条件および処理の違いを識別するための新たな番号「**ポリシーNo**」をタグＩＤと共に無線タグに付与する。

図６に処理イメージを示す。案Aは、サービスを実施するエリアのリーダ全てに設定しなくてはならない。また、設定を変更する負荷も大きい。案Bは、割り当てたＩＤの分だけ無線タグにメモリが必要になる。また、タグに対して処理が変わった時も、その設定を変更する負荷が大きい。案Cは、既存の無線タグＩＤに影響なく様々な処理が可能である（柔軟性が高い）。

以上から、案Cの採用が望ましい。ポリシーNo導入により、同じシステムＩＤでも場所や時間帯によって異なる処理、暗号化に対応することが可能になる。また、その仕組みは全システムで統一することができ、サービス開発も容易となる。

システムＩＤと同様に、ポリシーNoは無線タグ内に格納され、仮想リーダにより読み取られた後は仮想処理サーバまで引継がれる。各仮想化要素は、そのポリシーNoをもとに必要な処理を実施する。ポリシーNo

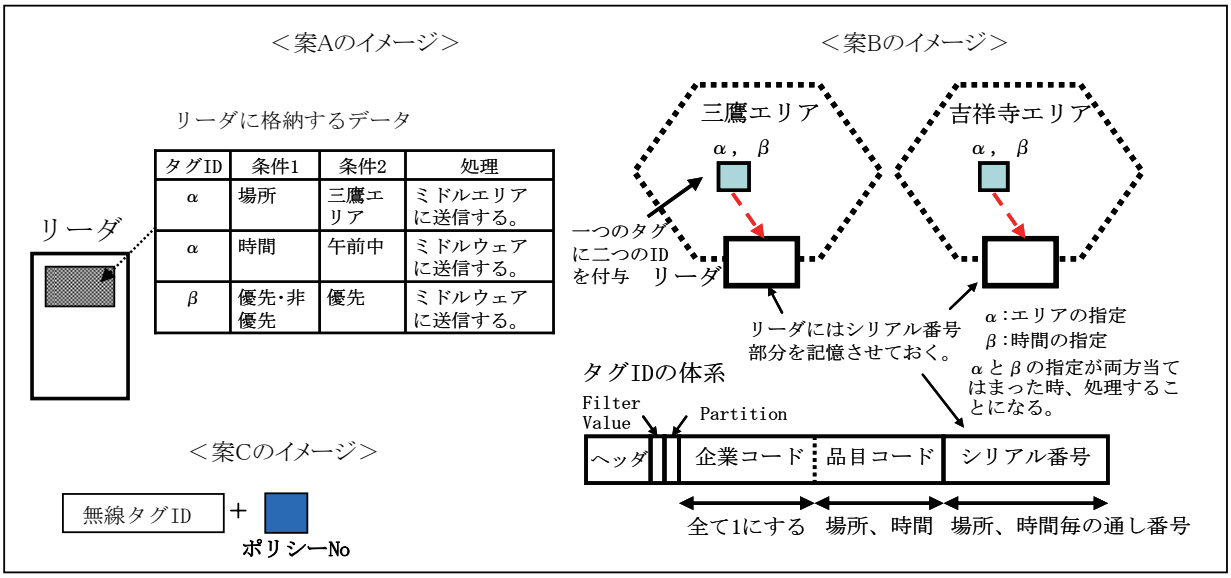
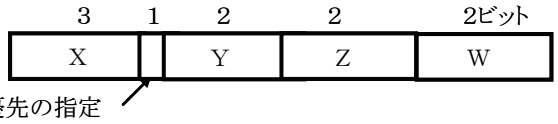


図6. ポリシーの実施方法



X:処理の大分類
000:無条件にミドルウェアに送信する。(デフォルト)
001:エリアを指定して、特別なアクションを要求する場合。
010:時間を指定して、特別なアクションを要求する場合。
011:エリアと時間の組み合わせ
100～:予約
W：暗号化の指定

Y:処理の指定（Aが001の場合）
00:Cで指定したエリアにいない場合に、ミドルウェアに送信する。
01:Cで指定したエリアにいる場合に、ミドルウェアに送信する。
10,11:予約

Z:処理の指定（Aが010の場合）
00:Cで指定した時間帯でなければ、ミドルウェアに送信する。
01:Cで指定した時間帯であれば、ミドルウェアに送信する。
10,11:予約

図7. ポリシーNoのコード体系例

コード体系案を図７に示す。

３プロセスマイグレーション技術

３−１研究の目標

仮想無線タグネットワークシステムの処理サーバの対故障性、対負荷変動性を強化するための強プロセスマイグレーション技術について検討する。

ネットワークを通じて高速で中断なくサービスを提供するために必要な技術の１つとして、サーバ機能およびサービスアプリケーションをネットワーク中の他コンピュータに移動させるプロセスマイグレーションがある。多くの処理サービスが処理サーバやネットワークに投入されると負荷がダイナミックに変動するため、それぞれのアプリケーションの並列化・分散化の仕方を適切に変えていかないと、高性能でかつ信頼性の高い処理は実現できない。その対応を個々のアプリケーションに組み込むのは決して容易ではない。マイグレーションの条件をシステムが自動的に認識し､かつユーザに負担を掛けずに移動できるようにするため、システムのベースとなる部分から自律性を持たせることが必要である｡そこで本研究では Java ベースのモバイルエージェントを用いるものとする。

モバイルエージェントのモビリティには、弱マイグレーションと強マイグレーションがある。弱マイグレーションはマイグレーション後にプログラムの初期状態からの再開となり、移動前の処理が無駄となる問題がある。そのため、高い自律性を実現するには強マイグレーションであることが望ましい。ただし、従来の強マイグレーション方式では、Java 仮想マシン（JavaVM）自体の変更を前提しており、様々なマシンが共存する環境で標準の JavaVM をそのままで利用するには困難な点がある。

本章は、強マイグレーションモバイルエージェントを容易に記述できる方法をユーザに提供し、そのエージェントコード標準の JavaVM 上で実行できる手法を確立することを目的とする。

３−２モバイルエージェントシステムと自律性

一般的な分散システムにおいては、タスクの負荷分散や停止したタスクの再生処理等の作業にはシステム全体の状況把握や管理が必要であり、それらのコントロールをすべて考慮してプログラム記述者がアプリケーションプログラムを記述するのは非常に困難である。そこで、あらかじめシステムに自律性を持たせることで、これらの問題を解決することができる。システムの自律性を満たすためには、システム内で動く各ソフトウェアが他からのメッセージに応じた行動と、自ら取得した外部の情報に応じて判断を行い、その結果によって行動を計画し、その計画に基づき行動できる必要がある。

エージェントとは、人間の代理としてシステム上で自律的に行動するソフトウェアであり、モバイルエージェントとはネットワークを通じてマシン間を移動しながらタスク処理を行うことができるエージェントのことを言う。また、モバイルエージェントシステムとは、モバイルエージェントの動作（生成、消去、移動、保存、複製、通信など）を提供するシステムのことである。既存のモバイルエージェントとしては、

AgentSpace（佐藤一郎氏）[11]やAglets（日本IBM社）[12]、JavaGO（東京大学米澤研究室）[13]、MOBA（首藤一幸氏）[14]等が挙げられる。

ネットワークに繋がれたコンピュータの性能は全て同じではなく、各コンピュータも課された処理を行うため性能は常に一定ではない。従って、仮に処理開始時直近における各コンピュータの性能値に合わせて自律的に各コンピュータへのエージェントの割り当てを調整したとしても、その後に追加される処理によって負荷は変動し、エージェントの処理の完了が大幅に延期される事態が生じてもエージェントはただ待つしかない。そこで、そのような場合に自動的に性能に余裕のあるコンピュータへタスクエージェントが移動し、大幅な処理の遅延を防ぐ機能が必要となる。

この機能実現には、エージェントが任意の中断した時点の実行時データを保持し、移動先のコンピュータ上で中断時点から処理を再開できなければならない。そこで中断時点での再開が可能な実行時データを持って移動する‘強マイグレーション方式’のモバイルエージェントシステムを構築する。

3-3 エージェントの強マイグレーション化

(1)　強マイグレーション化における問題点

Java においてプログラムが使用するメモリ領域には、実行コード領域（プログラムの実行コードを格納する領域)、スタック領域、ヒープ領域がある。このうち実行コード領域とヒープ領域については Java に備わっているシリアライズ機能を用いることによって保存が可能である。しかし、スタック領域内の情報やプログラムカウンタを実行状態として保存する機能は現状の JavaVM 1.6.0 ではサポートされていない。プログラムカウンタを取得・復元できれば移動直前の位置からのプログラムの再開が可能となる。また、スタック領域内に保持されているローカル変数の値を取得・復元できれば、移動前の計算結果を無駄にせず、実行を再開することができる。そのため強マイグレーション方式のモバイルエージェントシステムを実現するためには、スタック領域内の情報とプログラムカウンタに相当する情報が必要となる。

しかし、これを Java で実装するのは容易ではない。Java を用いた既存の強マイグレーション方式のモバイルエージェントとしては、JavaGoX[15]、MOBA、Nomads[16]、Brakes[17]などが挙げられるが、これらの研究では JavaVM 自体の変更を行ったり、バイトコード変換を行ったりしている。JavaVM を変更する場合、JavaVM のバージョンアップごとに修正を行わなければならず、システム開発の負担が大きい。

そこで本研究では、JavaVM に手を加えることなく強マイグレーション方式のモバイルエージェントシステムを実現する。しかも記述上は、プログラム中に単に移動命令を「migrate(移動先ホスト名);」のように記述するだけでマイグレーションを可能とし、この命令の続きから実行できるようにする。そのためには、スタック領域内のローカル変数の取得・復元、プログラムカウンタ相当の情報の取得・復元、が必要となる。その実現法を以下に説明する。

(2)　スタック領域の取得

スタック領域の中身を取得するために JPDA を使用する。JPDA は JavaVM に標準で実装されており、実行中のスレッドや実行情報へのアクセスが可能である[18]。

図８は、ローカル変数を取得し、持ち出し可能にするまでのプロセスの概要を示したものである。JPDA を用いてエージェントが実行されているスレッドにアクセスし、スタック内のローカル変数の値を取得する。しかし、このローカル変数の値は JPDA 特有のクラス(Value 型)であるため、外部への持ち出しが許可されていない。そこで、この値をシリアライズ（直列化）可能な変数に変換し、ヒープ領域内に格納する。これで JavaVM を変更することなく、Java に備わっているシリアライズ機能のみを用いて実行コード領域、ヒープ領域そしてスタック領域内のローカル変数を保存、別マシンに送信することができる。

(3)　スタック領域の復元

スタック領域を取得し別マシンに移動した後、移動先のマシンで実行を再開するには、マイグレーション

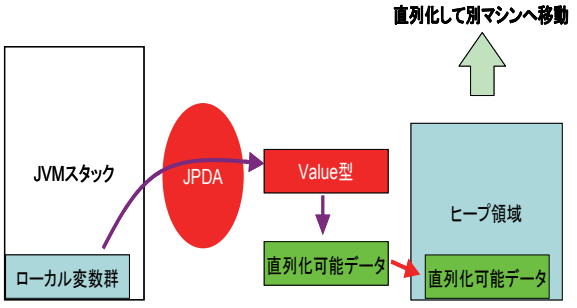


図８．ローカル変数持ち出しまでのプロセス

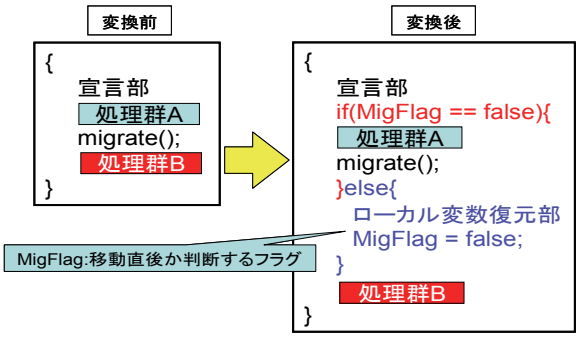


図９．　変換の基本方式

命令の直後までの状態を復元し、そこから実行を再開する必要がある。そのためにはスタックの中身だけでなく、プログラムカウンタも必要とされる。JPDA では、このプログラムカウンタに類似したデータを扱っており、これを取得することは可能である。しかし、この取得したプログラムカウンタを用いてプログラムを途中から実行する機能は、JPDA でもサポートされていない。そのため、ソースコードの変換を行うことにより、同等の機能をサポートすることにした。このソースコード変換手法では、プログラムコード中に記述された移動用関数である migrate メソッドの位置で実行時データを取得して移動し、到着後に移動直前の状態に復元するように変換する。

(4) ソースコード変換の基本形

ソースコード変換の手法は、プログラムコード中に記述された移動用関数である migrate メソッドの位置で実行時データを取得して移動し、到着後に移動直前の状態に復元するように変換する。この変換手法を実現するためにまず、エージェントが持って移動する情報の中に、マイグレーション直後かどうかを示すフラグ MigFlag を用意する。このフラグが true のときには移動直後であることを示す。初期値は false であり、エージェントの移動が行われると true になる。

この MigFlag と if～else 構文を組み合わせ､移動直後には実行再開位置までのスタックフレームを再構築し､移動前にすでに得られていた値についての計算処理は全てスキップするようにソースコードを変換する。この変換により、移動前のプログラムカウンタを保存して続きが実行できるのと同じ状態を実現することが可能となる。

具体的な変換作業は、移動関数である migrate()が存在するスコープを中心に考える。まず、その後の変換の前準備として、注目しているスコープ内で定義している宣言文をスコープの先頭にまとめて配置する。その結果として得られる「宣言部・処理群・migrate メソッド・処理群」という構造が変換対象の基本形となる。プログラム中で利用されるどのような構文でも一般的にこの基本形に直すことができる。図９は、この基本形から JavaVM 上での強マイグレーションコードへの変換方式を示している。ここで、ローカル変数の復元は、移動前の個々のローカル変数の値を移動後の対応するローカル変数に再代入するための一連の代入文をソースコードに挿入することにより実現する。

３－４ JavaCCを用いたソースコード変換の自動化

(1) 変数リスト、変数宣言行リストの作成

ソースコード変換の自動化にあたり、JavaCC(Java Compiler Compiler)[19]を使用した。JavaCC とは、lex/yacc を Java 言語に置き換えたものである。ユーザが記述した migrate メソッドが使用されている個数およびその位置によって変換後のソースコードの形が異なるので、ソースコードを１パスで解析し変換するのは困難である。そこで２パス方式とし、字句解析・構文解析を一度行って migrate の使用状況をはじめ必要な情報を取得し､その後もう一度ソースコードを解析して､最初に取得した情報を元にコード変換を行う。

構文解析フェーズで、変数のリスト、migrate メソッドの位置情報を取得する。同一スコープ内の変数宣言をスコープの先頭にまとめるためには、一度目の解析で、宣言されている変数のリストを作成する必要がある。そのリストに含まれる情報は変数の型と名前、宣言されている関数の名前、宣言されているスコープの深さなどである。

(2) migrate メソッドに関する情報の取得

正しいソースコード変換のため、宣言されている変数のリストと共に、使用されている migrate メソッドについて migrate が配置されている関数の名前とスコープの深さ、migrate メソッドの使用回数の情報を取得する。また、migrate メソッドが、ユーザが作成した関数内にある事も考えられるため、どの関数を呼び出しているかの情報も取得する必要がある。その理由は、migrate メソッドを呼び出すまでに終了している関数は、スタックフレームの復元を行う変換の必要が無いので、migrate に関与している関数の情報だけを取得し、その他の関数に対して不必要な変換を行わないようにするためである。

さらに、migrate メソッドが if 文や for 文といった制御文の中に記述されている場合もあるので、そのスコープ内で migrate メソッドが記述されている制御文に関する情報も取得する必要がある。この場合も migrate メソッドが記述されている制御文のみを変換し、その他の制御文に対して不必要な変換を行わないようにする。

(3) 自動ソースコード変換の具体操作

１パス目で取得した情報を元に、２パス目の解析でソースコード変換の基本形に則って変換を行う。変換の具体操作を以下に示す。

・プログラムファイル名は"Translated_" + オリジナルのファイル名とする。それに伴い、コード内のクラス宣言部分も変更する。

・migrate メソッド呼出が含まれるユーザ関数に到達したら、取得した変数リスト、変数宣言行リストを用いて、スコープの先頭に変数宣言をまとめる。

・変数宣言をまとめた後、その直後の行に AgentCoordinator を取得する命令文を挿入する。

・migrate 前後かどうかを判断する if 文を挿入する。

・migrate メソッド呼出までの処理郡を出力する。

・migrate メソッド呼出後、自身のスレッドを終了させる関数の呼出文を挿入する。この関数自体も自動で挿入する。

・migrate 後に変数復元を行うための else 文、変数復元部、MigFlag の初期化を行う関数呼出を挿入する。

3-5　提案した強マイグレーション方式の動作検証

３－３、３－４で提案した強マイグレーションモバイルエージェントを動作させるシステム AgentSphere を設計・実装し、動作確認を行った。AgentSphere を各コンピュータで起動することにより、エージェントをそれらのコンピュータに送り込むことができる。図１０のようなソースコードを用意し、このコードを自動変換した。図１０のプログラムは、５秒間 j の値をインクリメントし、その値を出力したら別のマシンへと移動を行う。また、i の値が指定した値になったら、別のマシンへ移動するというものである。これを２台のマシン間で行う。migrate メソッドはどのような制御構造でも対応できることを確認するため、ユーザ関数の中かつ for 文、if 文の制御文内に２つ記述してある。このソースコードを変換すると図１１のようなコードが出力される。表１は動作確認に使用したマシンを示す。２台のマシンで処理を行った時の、マシンＡでの結果を図１２に、マシンＢでの結果を図１３に示す。

これらの図から、変数の状態を保持しながらエージェントが移動し、移動後にその続きから処理を正しく行っていることが確認できる。

表１．　動作確認に使用したマシン一覧

マシン名	CPU名	クロック数	メモリ
マシンA	Core 2 Duo	2.2GHz	2.0GB
マシンB	PentiumD	2.8GHz	1.0GB

４ むすび

ユビキタスネットワーク社会の進展に伴い、物やオブジェクトを識別するための無線タグ利用が拡大し、それを用いた多種多様なサービスはインターネットのように国民生活になくてはならないものとなってくる。そのため、本研究では無線タグネットワークシステムの社会インフラ化に向け、仮想化を前提としたシステム共用化方式を明らかにした。具体的には、無線タグ以外の装置は従来のコンピュータやネットワークの仮想化技術を組み合わせることにより実現できることを明らかにした。仮想化は装置の共用化によりコストダウンだけではなく、特にリーダなどの設置スペースや運用管理稼動の大幅な削減も期待できる。また、仮想化が困難な無線タグと他仮想化要素の連携のため、システムやサービスを識別する専用のＩＤ（システムＩＤ）と条件や処理の違いを明記した専用の番号（ポリシーNo）を従来のタグＩＤとは別に無線タグ内に格納する方式を提案し、その具体的な実現法を明らかにした。

さらに、処理サーバやネットワークの故障時またはふくそう時にも仮想無線タグネットワークシステムをサービス中断なく運用するために必要な強プロセスマイグレーション方式を明らかにした。具体的には、モバイルエージェント方式を前提に、ユーザにとって簡単なエージェントの記述法（ユーザがプログラムソースコードの任意の位置で migrate 命令を記述できる）ならびにそのコードを標準の JavaVM で実行するための自動ソースコード変換方式を明らかにした。実機を用いた動作検証を行い、提案方式の有効性を明らかにした。

```
public class TransCode extends StrongMigAgent
implements Serializable {
  private String IPaddress
  private static AgentCoordinator userAC;

  public void create(){
    int a = 20;
    func1();
  }

  public void func1(){
    int i=10,j=0;
    long start,end,sum;
    System.out.println("start!");
    for(int a=0; a<10; a++){
      start = System.currentTimeMillis();
      j=0;
      while(true){
        j=j+1;
        end = System.currentTimeMillis();
        if((end-start)/1000 > 5)  break;
      }
      if(a%2==0) IPaddress = "133.220.114.127";
      else  IPaddress = "133.220.114.240";
      i= i+1;
      System.out.println("a:" + a + " i:" + i + " j:" + j);
      userAC.migrate(IPaddress, 0);

      if(i==15){
        if(a%2==0)  IPaddress = "133.220.114.240";
        else  IPaddress = "133.220.114.127";
        userAC.migrate(IPaddress, 1);
      }
    }
    System.out.println("finish!!");
  }
}
```

図 10.　動作確認に用いたソースコードの一例

```
コマンド プロンプト - exe
[AgentSphere]> load Translated_TransCode.class
> load Translated_TransCode.class
LoadAgent : Translated_TransCode.class
OK
[AgentSphere]> start!
a:0 i:11 j:107217203
dispatch: Address 133.220.114.127 port 60000
a:2 i:13 j:107063422
dispatch: Address 133.220.114.127 port 60000
a:4 i:15 j:106380543
dispatch: Address 133.220.114.127 port 60000
a:5 i:16 j:106395541
dispatch: Address 133.220.114.240 port 60000
a:6 i:17 j:106600873
dispatch: Address 133.220.114.127 port 60000
a:8 i:19 j:106574569
dispatch: Address 133.220.114.127 port 60000
finish!!
j:45028825
i:20
create:a->20
エージェント終了！
```

図 12. マシンA

```
コマンド プロンプト - exe
MachineList表示
Machine No.0    :SEL27/169.254.0.1
Machine Perfo:1350005
Machine No.1    :/5.161.20.146
Machine Perfo:2412426


[AgentSphere]>
[AgentSphere]> a:1 i:12 j:44126520
dispatch: Address 133.220.114.240 port 60000
a:3 i:14 j:44163856
dispatch: Address 133.220.114.240 port 60000
dispatch: Address 133.220.114.240 port 60000
a:7 i:18 j:45068821
dispatch: Address 133.220.114.240 port 60000
a:9 i:20 j:45028825
dispatch: Address 133.220.114.240 port 60000
```

図 13. マシンB

```
public class Translated_TransCode extends
StrongMigAgent implements Serializable {
  private String IPaddress;
  int count=0;
  private static AgentCoordinator userAC;

  public void create(){
    int a = 20;
    userAC = getAgentCoordinator();
    if(userAC.checkflag(0)==false &&
            userAC.checkflag(1)==false){
    }else{
       a = userAC.getIntegerValue("a",1);
    }
      func1();
  }
public void func1(){
  int i=10,j=0;
  long start,end,sum;
  if(userAC.checkflag(0)==false &&
            userAC.checkflag(1)==false){
     System.out.println("start!");
  }else{
       //変数復元部
       i = userAC.getIntegerValue("i",0);
       ……
  }
   for(int a=0;userAC.checkflag(0)==true ||
            userAC.checkflag(1)==true ||  a<10; a++){
     if(userAC.checkflag(1)== false){
         if(userAC.checkflag(0)== false){
             start = System.currentTimeMillis();
             j=0;
             while(true){
                j=j+1;
               end = System.currentTimeMillis();
               if((end-start)/1000 > 5) break;
             }
             if(a%2==0)  IPaddress = "133.220.114.127";
             else   IPaddress = "133.220.114.240";
              i= i+1;
             System.out.println("a:" + a + " i:" + i + " j:" + j);
             userAC.migrate(IPaddress, 0);
         }else{
             //変数復元部
             userAC.setFlag(0);
         }
      }else{
         //変数復元部
      }
     if(userAC.checkflag(1)==true || i==15){
         if(userAC.checkflag(1)== false){
            if(a%2==0)  IPaddress = "133.220.114.240";
            else  IPaddress = "133.220.114.127";
            userAC.migrate(IPaddress, 1);
         }else{
            //変数復元部
           userAC.setFlag(1);
         }
     }
   }
   System.out.println("finish!!");
  }
}
```

図 11.　ソースコード変換後のコード

参考文献

[1] 経済産業省情報経済課：“電子タグ（ICタグ）の普及に向けた日本の戦略”,2004.7
http://www.slrc.kyushu-u.ac.jp/japanese/information/workshop/workshop02.pdf
[2] ユビキタスネットワーキングフォーラム　電子タグ高度利活用部会：“ベストプラクティクス集I～IV”
http://www.ubiquitous-forum.jp/documents/index.html
[3]“無線ICタグ活用のすべて”、日経RFIDテクノノロジ著、日経BP社
[4] 家電製品協会、日本自動認識システム協会、富士総合研究所“家電業界における無線タグの利活用モデルの実証実験　事業概要報告試料”H16.7.12http://www.mizuho-ir.co.jp/society/ictag/200400712h15.pdf
[5] EPCglobal, “The EPC global architecture framework Ver.1”
[6] ユビキタスIDセンタ http://www.uidcenter.org/japanese.html
[7] ユビキタスコミュニケータのマルチ通信インタフェース　http://www.uidcenter.org/pdf/barcode.pdf
など
[8]　畠山､津村､栗林:"無線タグネットワークの社会インフラ化に向けたシステム仮想化技術の検討", 2007年電子情報通信学会総合大会B-7-124.
[9] IETF RFC2547 “BGP/MPLS VPN” (March 1999)ＶＰＮ
[10] VMware Infrastructure (http://www.networld.co.jp/vmware/ infrastructure/main.htm)など
[11] 佐藤一郎：“AgentSpace：モバイルエージェントシステム”,日本ソフトウェア科学会, Dec.1998
[12] 日本 IBM 東京基礎研究所： http://www.trl.ibm.com/aglets/
[13] 米澤、関口、橋本：“移動コード技術に基づくモバイルソフトウェア”
http://homepage.mac.com/t.sekiguchi/javago/index-j.html
[14] 首藤一幸： http://www.shudo.net/moba/ ,Jan.2008 参照可
[15] Sakamoto, T., Sekiguchi, T., Yonezawa, A.: “Byte Code Transformation for Portable Thread Migration in Java”, In Proceedings of ASAMA’2000, Springer, Zuerich, Germany (2000) 16-28.
[16] Suri, N. et al.: “Strong Mobility and Fine-Grained Resource Control in NOMADS”, In Proceedings of ASAMA’2000, Springer, Zuerich, Germany (2000) 2-15.
[17] Truyen, E. et al.: “Portable Support for Transparent Thread Migration in Java”, In Proceedings of ASAMA’2000, Springer, Zuerich, Germany (2000) 29-43
[18] Illmann, T. et al.: “Transparent Migration of Mobile Agents Using the Java Platform Debugger Architecture”, Mobile Agents: Proceedings of the 5th International Conference, Ma 2001 Atlanta, Ga, December 2-4, pp.198- 212, 2001
[19] https://javacc.dev.java.net/, Jan.2008参照可

〈発　表　資　料〉

題　　名	掲載誌・学会名等	発表年月
無線タグネットワークシステムの社会インフラ化に向けたシステム仮想化技術の検討	電子情報通信学会 情報ネットワーク研究会 IN2007-70	2007 年 9 月
A Code Transformation Method For Strong Migration Mobile Agent	Pacrim’07 , F22-5	2007 年 8 月