2005－DBS－135　(7)
2005／1／20

Ja-Net on Grid における信頼できないホストのリソース利用を考慮したモバイルエージェントのセキュリティ機構の設計と実装

沼田哲史[†1]　板生知子[†2]　小川剛史[†3]
塚本昌彦[†4]　西尾章治郎[†1]

†1 大阪大学大学院情報科学研究科　†2 NTT 未来ねっと研究所
†3 大阪大学サイバーメディアセンター　†4 神戸大学工学部電気電子工学科

Ja-Net on Grid は，動的に変化するグリッド環境における並列計算のためのモバイルエージェントシステムである．本システムではこれまで Globus Toolkit のセキュリティ基盤 GSI を用いて，信頼できるエージェントのみをホストが受け入れられるようにセキュリティを構築してきた．しかし，エージェント側のセキュリティは考慮していなかったため，悪意のあるホストにエージェントの内部変数を不正に書き換えられるなど，エージェントがホストに攻撃される可能性があった．そこで本稿では，信頼できないホスト上であっても，モバイルエージェントが一定のセキュリティを確保し，グリッド上の全てのホストで処理を継続できる手法を提案する．

The Design and Implementation of Migration Method of Mobile Agents on Ja-Net on Grid for Increasing Reliability of Host Utilization

Satoshi Numata,[†1] Tomoko Itao,[†2] Takefumi Ogawa,[†3]
Masahiko Tsukamoto[†4] and Shojiro Nishio[†1]

†1 Grad. Sch of Info. Sci. and Tech., Osaka Univ.　†2 NTT Network Innovation Laboratories
†3 Cybermedia Center, Osaka University　†4 Dept. of Elec. and Elec. Eng. Faculty of Eng., Kobe Univ.

Ja-Net on Grid is a mobile agent system, which simplifies parallel calculations on dynamic changing grid environments. We have established its security by using a security infrastructure of Globus Toolkit, with which mobile agent hosts can accept only reliable agents. However, malicious hosts can rewrite instance variables of agents or extract important information from agents, since the security for agents has not been considered. We can prevent agents from migrating to unreliable hosts to keep the agents' security, though many of the resources connected on grid environment will be unavailable in the case. Thus, we propose a method that enables mobile agents to migrate securely onto unreliable hosts and continue their works even on unreliable hosts. In this paper, we describe the detail of the proposing method and its performance evaluation.

1. はじめに

我々の研究グループではこれまで，実空間に設置したカメラやセンサなどのデバイスを利用して大量のデータを収集し，それらのデータをネットワーク上の高度な計算機資源を活用して処理するための高度ユビキタス環境[9]の実現を目指して，グリッド上で自律的かつ適応的に動作するモバイルエージェントをベースにしたアプリケーションフレームワーク Ja-Net on Grid[11] を提案し実装してきた．Ja-Net on Grid は，エージェントがグリッド環境におけるネットワークのパフォーマンス変化などに動的に対応して効率的に処理を行うためのモバイルエージェントシステムである．

Ja-Net on Grid では Globus Toolkit[7] の GSI (Grid Security Infrastructure) を使用しており，それぞれのホストが実行を許可するエージェントを適切に識別してセキュリティを確保している．しかし，Grid 環境では膨大な計算リソースを利用できる一方，その中に悪意のあるホストがいる可能性もあり，エージェントの内部変数が不正に参照されたり，改竄されたりする可能性がある．信頼できないホストへの移動を禁止することでそれらの悪意のあるホストからエージェ

ントを守ることが可能であるが，Grid 環境における膨大な計算リソースを最大限に利用することができない．グリッド環境では，SETI@home[5] のように任意のユーザが貸し出しているリソースを有効に利用することも目的としており [10]，信頼できないホストを介しての移動を繰り返しても，安全にタスクを処理できることは重要である．

本稿では，Ja-Net on Grid において，信頼できないホスト上であってもエージェントが安全にタスクを実行するための手法を提案する．具体的には，エージェントがホスト間を移動する際に，エージェントのもつ変数を暗号化して，移動先のホストでその変数を安全に復号化して使用することによって，ホストからの不正な参照や改竄を防止する．また本手法を利用する場合に必要な処理コストを調べ，本手法の有効性を確認した．

以下，2 章では Ja-Net on Grid の概略とその問題点について述べる．3 章では関連研究について概要をまとめ，本稿における問題点への適用の可能性について述べる．4 章ではエージェントを保護する手法を提案し，その詳細について述べる．5 章では，提案手法の実装についてその詳細を述べる．6 章では，本手法の実現にかかるコストを分析してその有効性について考察する．7 章では本稿のまとめと今後の課題について述べる．

2. Ja-Net on Grid

本章では，Ja-Net on Grid のグリッド環境における動作の仕組みと現在のセキュリティについて述べ，その問題点について述べる．

Ja-Net on Grid では，サイバーエンティティ (Cyber Entity, CE) と呼ぶ自律的なサービスコンポーネント (モバイルエージェント) がランタイムシステムである ACERE (Abstract Cyber Entity Runtime Entity) の上でタスクを実行する．CE は ACERE から提供されるリソースが減少すると，他のリソースが潤沢な ACERE に移動して実行を続けることができる．各 CE は UUID による識別子をもっており，その識別子を使用して CE 間で通信を行う通信内容に応じて，自動または手動で CE 間にリレーションシップを生成することができる．具体的には，CE が内部にもつリストに，通信相手の CE がもつ UUID と通信の内容をキーとして，キー=バリューの組を生成する．CE はリレーションシップに格納されたデータを元に通信を行う相手を決定し，また移動時には，リレーションシップによって特定の関係をもった CE と同時に移動することが可能であり，協調動作を行う複数の CE に移動先でもその協調動作を続けさせることによって，効率的な実行が可能となる（**図 1**）．

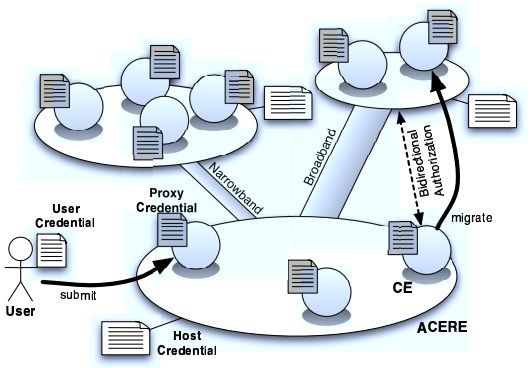

図 1 Ja-Net on Grid の概念図
Fig. 1 Overview of Ja-Net on Grid

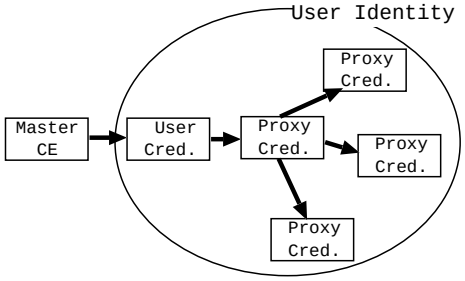

図 2 Ja-Net on Grid の認証動作
Fig. 2 Authentication of Ja-Net on Grid

図 2 に示すように，CE はそれぞれ，その CE を生成したユーザの証明書から派生したプロキシ証明書を持ち運ぶ．移動してきた CE のプロキシ証明書を参照することで，各 ACERE は実行を許可しているユーザが生成した CE だけを選択して動作させることが可能となる．

現在は信頼できる ACERE に接続されている ACERE はすべて信頼できるものとして移動先の決定を行っているが，これでは 1 章で述べたような，任意に貸し出されている膨大な計算リソースの安全な利用ができないため，任意の ACERE において CE が自分のセキュリティを確保できるような手段が必要となる．

3. 関 連 研 究

悪意のあるホストからエージェントを保護する手法について，いくつかの研究が行われている [3]．

3.1 部分結果のカプセル化

モバイルエージェントを用いて構築したオークショ

ンシステムでは，エージェントが各ホストを巡回して入札情報を収集し，落札者を決定する．このとき，あるホストが他のホストの入札情報を改竄してしまうと，正常な入札が行えない．そこで，文献 1) では，各ホストに独自の証明書を保持させ，その証明書を用いて各ホストが処理した変数に署名するようにしている．このようにすることで，他のホストが処理した情報の不正な参照や改竄が不可能となり，エージェントのタスクが正常に行われる．この手法を利用することで，エージェントを保護し，処理結果の信頼性を高めることが可能であるが，事前に信頼できるホストに証明書を配布しなければならず，信頼できないホストのリソースを活用したタスクの実行は実現できない．

3.2 実行の追跡

文献 2) では，部分結果のカプセル化と同様に，各ホストに独自の証明書を保持させ，それぞれのホストにおいて移動時の状態を署名付きで記録しておくことにより，どのホストにおいて改竄があったのかを検出している．この手法により，信頼できないホスト間を移動しても改竄があった場合にはそれを検出できるようになる．しかしこの手法では悪意のあるホストからのエージェントに対する不正なアクセスを防ぐことはできないため，本稿における目的は達成できない．

3.3 関数の暗号化

文献 4) では，実行対象となるメソッドを移動前に変換し，移動先のホストに意味のある値や演算を参照できないようにすることでエージェントの動作の保護を試みている．ホストに戻った後，そのエージェントは計算結果に対して逆の変換を行い，意味のある値を結果として取り出す．我々の提案手法では鍵を用いて変数を暗号化することでエージェントの保護を試みるが，この手法では関数の動作を暗号化することでエージェントの保護を試みている．

3.4 Locker パターン

Locker パターン [8] では，ホスト上にロッカーと呼ぶオブジェクトの保管庫を用意し，エージェントが他のホストに移動する際に，セキュリティ上問題があると思われるすべての情報をそのロッカーに預けてから移動する．これにより，エージェントがたとえ悪意のあるホストに移動しても，重要な情報に不正にアクセスされることはない．しかし，ロッカーに入れてしまった情報に関するタスクは，他のホストに移動すると継続できない．

4. CE の保護

本章では，移動先の ACERE による変数の覗き見や改竄が不可能な CE の保護手法を提案する．

4.1 提 案 手 法

Ja-Net on Grid のこれまでの実装では ACERE が Java のリフレクション API を用いることで CE の内部変数を覗き見たり改竄したりすることが可能である．本稿ではこの問題に対処する方法について述べる．なお，バイトコード自体や直列化されたバイト列が改竄されることはないものとする．

まず考えられる簡単な対処方法は，CE の変数を private なメンバとして宣言しておくことである．これにより，デフォルトの環境ではリフレクション API を使用しても外部からこの変数にアクセスすることができなくなる．しかし，セキュリティマネージャがない場合や，ReflectPermission(”suppressAccessChecks”) に対応した checkPermission() メソッドが SecurityException 例外をスローしないセキュリティマネージャを用意している場合には，その変数の Field クラスの setAccessible() メソッドを true を引数にしてコールすることでアクセスできるようになる [6]．そのため，CE の変数を private なメンバとして変数を宣言しても，ACERE から簡単に覗き見たり改竄したりすることが可能であり，完全に CE を悪意のある ACERE から保護することはできない．

信頼できる ACERE のリストを事前に CE に与えておき，CE は移動する際，その候補をリストから選択するようにすれば CE の安全性を保持できるが，利用する ACERE を限定してしまい，グリッド環境に接続された多くのリソースを利用できなくなってしまう．本稿では，一定のセキュリティを保った上で，ネットワーク上のすべての ACERE を利用して計算できるようにすることが目的である．

提案手法では，CE のもつすべての変数を暗号化し，CE がタスクを行う際にのみ復号化してデータを参照・更新できるようにする．具体的な処理手順を以下に示す．

(1) **変数の暗号化**　CE が移動する際に，移動元の ACERE が鍵を生成する．CE は，その鍵を使用して CE のもつすべての変数を暗号化する．
(2) **鍵の取得**　CE は移動先で移動元の ACERE とソケット通信を行い，鍵を受け取る．
(3) **変数の参照と更新**　鍵は処理が終了するまで各メソッドに引数として渡され，CE は変数を参照するときと変数を書き換える場合にこの鍵を使用して復号化・暗号

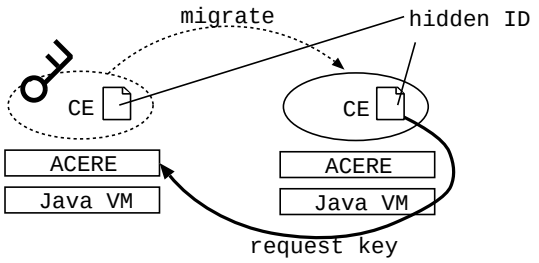

図 3 ACERE を介さない鍵の取得
Fig. 3 Key acquisition not through ACERE

```
class MyCE extends AbstractCyberEntity {
  public MyObj foo;
  public void work() {
    MyObj foo2 = /* foo を用いた操作 */;
    foo = foo2;
  }
}
```

図 4 変換元の CE
Fig. 4 Source Class

化を行う．

(4) 変数の復号化 タスクの処理を完了した後，CE は移動元の ACERE に戻って変数を復号化する．

手順 (2) におけるソケット通信を Java API を使用して行うことで，ACERE は通信に介入できなくなる（**図 3**）．移動する CE のクラスには，その CE に対応した外部に公開されない ID が埋め込まれる．そして移動後に CE が鍵を取得する際に，自分の移動元の ACERE に送る．移動元の ACERE は鍵の取得要求が CE からのものであることを確認した上で鍵を送信する．この ID は定数としてもつのではなく，鍵を取得するためのメソッド内で一時的に参照する値としてもつようにする．そのためこの値はコンスタント・プールに格納されるが，Java のリフレクション API では参照することはできない．CE が受け取った鍵はヒープ上に格納され，ACERE からは リフレクション API を使用しても鍵にアクセスできない．また復号化された値は常にヒープ上にのみ格納する．メンバ変数の値の更新はその値を暗号化してから元の値と入れ替えるため，変数は常に暗号化された状態となって，ACERE からはアクセスすることができなくなる．以上のように，本手法では CE のメンバ変数を暗号化し，他の ACERE に移動した後も，ACERE がアクセスできない領域に必要なデータを保持することによって，CE の安全性を確保している．

本手法は，従来の Ja-Net on Grid システム用に構

```
class MyCE extends AbstractCyberEntity {
  public MyObj foo;
  public Map variableMap;
  private Object getVariable(String name,
      Key key) {
    byte[] encryptedBytes =
  (byte[]) variableMap.get(name);
    byte[] decryptedBytes =
      /* 暗号化したバイト列を key で複合化 */;
    Object obj = /* 復号化したバイト列から
      オブジェクトを復元 */;
    return obj;
  }
  private Object setVariable(String name,
      Key key, Object obj) {
    byte[] bytes =
      /* obj を直列化したバイト列 */
    byte[] encryptedBytes =
      /* バイト列を key を用いて暗号化 */
    variableMap.put(name, encryptedBytes);
  }
  public void work() {
    Key key = /* ソケット通信で鍵を取得 */;
    work_(key);
  }
  public void work_(Key key) {
    MyObj foo_=(MyObj)getVariable("foo",key);
    /* foo_ を用いた操作 */
    setVariable("foo", key, );
  }
}
```

図 5 変換後の CE
Fig. 5 Modified CE

築されたアプリケーションのソースコードを書き換えることなく適用することが可能であり，本手法を用いることで，各 CE は自分の状態を盗み見られたり改竄されたりすることなく任意の ACERE 上でタスクを処理できるようになる．

5. 実　　装

本節では，Ja-Net on Grid における提案手法の実装について述べる．

5.1 バイトコードの変換

本手法は，CE のバイトコードを変換することで実現できる．まず保護すべき変数を参照しているすべてのコードと変数に値を代入するすべてのコードを，それぞれ鍵を使用して値を複合化するメソッドと暗号化するメソッドの呼び出しに変換する．暗号化は，元のオブジェクト（int 型などのプリミティブ値の場合にはそれに対応する Integer クラスなどのオブジェクト）を直列化してバイト列に変換した後，そのバイト列に対して暗号化を行い，暗号化されたバイト列を取

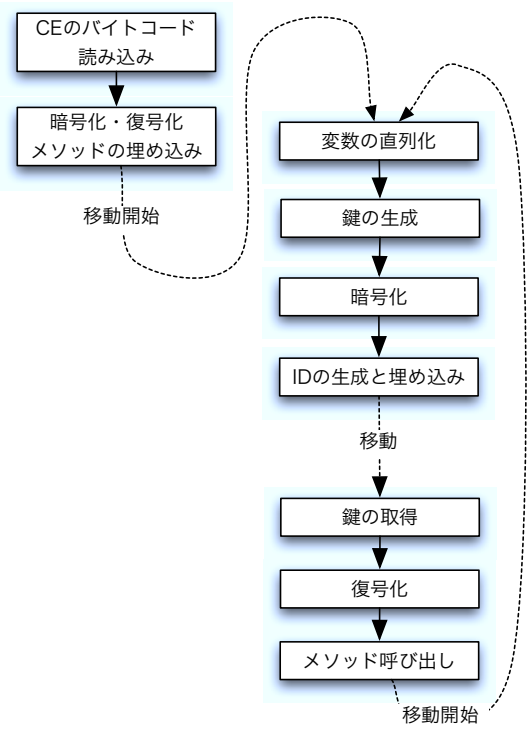

図 6 暗号化を用いた実行手順
Fig. 6 Execution with Encryption

得することで行われる．暗号化されたバイト列の格納先として，Map オブジェクトが 1 つ用意される．ここに変数名をキーとして暗号化されたバイト列が格納される．変数を参照するためのメソッド getVariable() はこのバイト列から暗号化されたバイト列を読み出して復号化を行い，変数を更新するためのメソッド setVariable() は新しい値を暗号化したバイト列に変換してこの Map に格納し直す．たとえば**図 4** のような CE が与えられた場合，これは **図 5** のように変換される．

処理手順は，**図 6** に示すように，ACERE が CE のバイトコードを読み込んだ時点で，暗号化と復号化に必要なメソッドを埋め込み，既存のメソッドが鍵を使用して変数の暗号化・復号化を行うように変換する．そして CE は移動する際にすべての変数を直列化し，その直列化したバイト列を暗号化してから，ネットワーク越しに鍵を取得するために必要な ID を埋め込み移動を開始する．

5.2 変数の直列化と暗号化

CE で扱うオブジェクト変数はすべて java.io.Serializable インタフェースを実装することとしており，バイト列への直列化は，java.io.ObjectOutputStream クラスと java.io.ByteArrayOutputStream クラスを使用して行う．バイト列からオブジェクトへの復号化はその逆で，java.io.ObjectInputStream クラスと java.io.ByteArrayInputStream クラスを使用する．int 型などのプリミティブ変数の場合には，Integer クラスなど，対応する型のオブジェクトに変換してからバイト列に変換する．

```
public byte[] encryptObject(
  SecretKey key, Object object)
{
  Cipher cipher = Cipher.getInstance(
    key.getAlgorithm());
  cipher.init(Cipher.ENCRYPT_MODE, key);
  PipedOutputStream pos =
    new PipedOutputStream();
  PipedInputStream pis =
    new PipedInputStream(pos);
  ObjectOutputStream oos =
    new ObjectOutputStream(pos);
  oos.writeObject(object);
  oos.flush();
  ByteArrayOutputStream baos =
    new ByteArrayOutputStream();
  while (pis.available() > 0) {
    baos.write(pis.read());
  }
  return cipher.doFinal(baos.toByteArray());
}

public Object decryptObject(
  SecretKey key, byte[] bytes)
{
  Cipher cipher = Cipher.getInstance(
    key.getAlgorithm());
  cipher.init(Cipher.DECRYPT_MODE, key);
  byte[] decBytes = cipher.doFinal(bytes);
  return createObjectFromBytes(decBytes);
}
```

図 7 オブジェクトの暗号化・復号化メソッド
Fig. 7 Encryption/Decryption method for Objects

暗号化には，共通鍵暗号アルゴリズムである DES (Data Encryption Standard) を用いて生成した，javax.crypto.SecretKey インタフェースを実装している鍵を用いている．上記のようにして直列化したバイト列を，鍵を元に生成した javax.crypto.Cipher クラスのインスタンスを用いて暗号化することで，暗号化されたバイト列が得られる．復号化も同様に行う．**図 7** に，オブジェクトの直列化と暗号化・復号化を行う encryptObject() メソッドと decryptObject() メソッドの実装を示す．

5.3 ID の埋め込み

ネットワーク越しに鍵を取得するために必要な ID は，コンスタント・プールに埋め込む．コンスタント・

プールは，オブジェクトの情報を含むバイトコードの一部であり，Ja-Net on Grid では，すべての CE が移動時に CE 自身のバイトコードと状態を移動先の ACERE に送信する．本手法では，このバイトコード自体は改竄されないことを前提としている．CE は移動先の ACERE から移動元の ACERE に SSL を用いたソケット通信でその ID を送信することで，正式な鍵を取得する．このソケット通信は ACERE から覗き見ることはできず，鍵は CE のスタック上に格納される．ひとつのスレッドに対応したスタック上のこの鍵は，ヒープ上に格納されるローカル変数とは異なり，Java の機構によって CE 以外から参照することはできない．そのため，悪意をもったホストが不正な Java VM を用意しない限り鍵が盗まれることはなく，変数が改竄されたり覗き見られたりする危険性を回避することが可能となる．

なお，鍵の取得のための ID をコンスタント・プールに埋め込んでいるのと同様に，コンスタント・プールに直接鍵を埋め込むことも可能であるが，バイトコードの解析と比較してコンスタント・プールの参照は容易であるため，直接埋め込むようにはしていない．移動元の ACERE から移動先の ACERE に対してソケット通信を通じて鍵を送るのは 1 度だけとしており，ダミーの ID をコンスタント・プールに混在させることでセキュリティを高めることができる．またその ID をさらに何らかの手法（バイト列の並びを変える，定数を加算するなどの操作をランダムに組み合わせる，など）で変換しておいて ID を利用する際に復元するメソッドを介するようにすれば，バイトコードの参照なしに鍵を取得することができなくなり，試行を繰り返して鍵を復元することはできなくなるため，さらにセキュリティを高めることができる．ACERE はコンスタント・プールからランダムに抜き出した値を使用して鍵の要求を繰り返すことで一定数の CE に対して不正なアクセスを行うことが可能であるが，正しい ID を伴わない不正な鍵の要求を繰り返す ACERE は想定している以上に危険な ACERE であると判断して，その利用を中断することが可能である．

5.4 ACERE への影響の考慮

本手法では，ACERE を介さずに直接 Java API にアクセスすることでヒープ上に鍵を取得し，ACERE からその鍵を参照されることを防いでいる．これは ACERE にとってはセキュリティ上好ましくないが，CE のソケット通信は特定のポートでのみ行うものとしておけば，CE が作成するソケット通信用の java.net.Socket オブジェクトが，移動元のホスト名および鍵取得用のポート番号の定数でのみ初期化されることをバイトコードレベルで確認することが可能であるため，そのようなライブラリを ACERE に提供しておくことで，ACERE は不用意にポートを空けられ，DOS 攻撃などに利用される心配はなくなる．

6. 性 能 評 価

本手法を適用した場合に発生するコストは，暗号化・復号化のコストと鍵を送るためのコストである．本章では，提案手法を実現する際のそれぞれのコストを分析し，この手法を実際に適用した場合のパフォーマンスについて考察する．

6.1 実 験 環 境

性能評価のための実験には，次のような構成の 2 台のマシンを使用した．

Linux マシン
- CPU: Intel Pentium III 1.4GHz
- HDD: 120GB
- メモリ: 512MB
- OS: RedHat Linux 7.3
- JDK: JDK 1.4.1_03-b02

PowerBook G4
- CPU: PowerPC G4 800MHz
- HDD: 60GB
- メモリ: 1GB
- OS: Mac OS X 10.3.5
- JDK: JDK 1.4.2_05-141

6.2 暗号化・復号化のコスト

6 文字のアルファベットからなる Java の文字列オブジェクト，すべての CE がもつ WorkingRange オブジェクト，そして int 型のプリミティブ値を表すための Java の Integer オブジェクトの暗号化と復号化にかかるコストを計測した．暗号化の対象はオブジェクトをシリアライズして得られたバイト列であり，それぞれのバイト列のサイズは，文字列オブジェクトが 12 バイト，WorkingRange オブジェクトが 61 バイト，Integer オブジェクトが 81 バイトである．

図 8 と**図 9** に Linux マシンにおける暗号化および復号化のコストを，**図 10** と**図 11** に PowerBook G4 における暗号化および復号化のコストを示す．横軸は連続して暗号化あるいは復号化を行った回数であり，縦軸は総所要時間を繰り返し回数で割って算出した，オブジェクト 1 個あたりの暗号化あるいは復号化にか

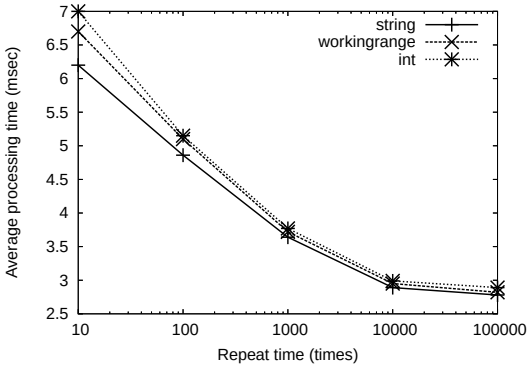

図 8 Linux マシンにおける暗号化のコスト
Fig. 8 Cost of encryption on Linux machine

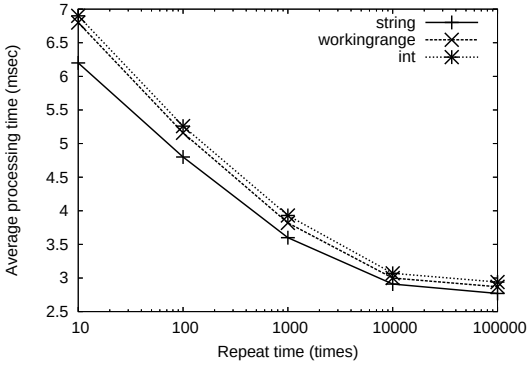

図 9 Linux マシンにおける復号化のコスト
Fig. 9 Cost of decryption on Linux machine

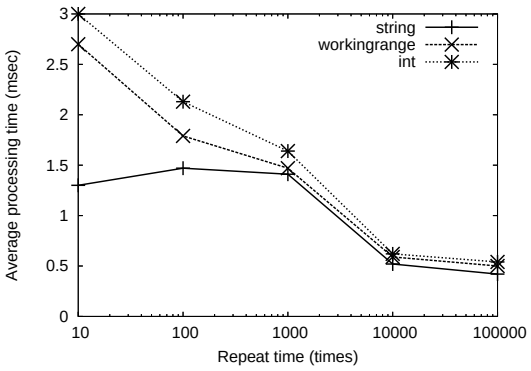

図 10 PowerBook G4 における暗号化のコスト
Fig. 10 Cost of encryption on PowerBook G4

かった時間である．

この結果より，Linux マシンにおいても PowerBook G4 においても，暗号化と復号化のコストはほぼ同じである．PowerBook G4 の方がプロセッサ速度が 2 倍近い Linux マシンよりも 2〜3 倍速く暗号化と復号化の処理を終えているが，Linux と Mac OS X における Java VM の最適化処理，暗号化・復号化処理の実装の違いによるものである．

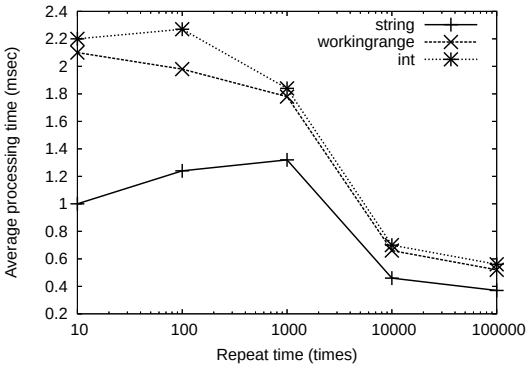

図 11 PowerBook G4 における復号化のコスト
Fig. 11 Cost of decryption on PowerBook G4

実際にユーザが実装する CE のもつ変数の数は 10 個〜100 個程度であると考えられるため，暗号化および復号化にかかるコストは，Linux マシンの場合でおよそ 5〜7 msec，PowerBook G4 の場合でおよそ 1.5〜3 msec であると考えられる．これより，800MHz 以上のプロセッサ速度をもつマシンを使用した場合には，10 個の変数をもつ CE の場合におよそ 10.5〜70 msec，100 個の変数をもつ CE の場合におよそ 105〜700 msec のコストが生じることとなる．分散処理させた場合には，移動距離が 1 ホップ伸びるごとにこのコストがホップ数分かかってくるため，何ホップの移動を許可するかによってリニアにコストが加算される．1 ホップの場合には，700 msec が最悪のコストである．実際のアプリケーションにおいてはこのコストと，次に示す鍵送信のコストが加算されたものが本手法を適用する際に全体としてかかるコストになるものと考えられる．

6.3 鍵送信のコスト

鍵を送信するためのコストを計測するため，802.11b の無線 LAN（最大 11Mbps）と 1000BASE-T のギガビットイーサネットワーク（最大 1Gbps）の 2 種類のネットワークを用意し，それぞれのネットワークにおいて鍵の送付にかかる時間を計測した．この計測は，ACERE から CE への鍵データの送信を開始した時点から，鍵の受信が完了したことを知らせるために CE から ACERE へ送られる 1 バイトのデータを ACERE が受け取るまでの時間を計測している．その結果，無線 LAN 上では平均 7.33 msec，ギガビットイーサネットワーク上では平均 2.67 msec で鍵の送付が完了することが分かった．これより，本手法を適用した場合のパフォーマンスの低下は，1 ホップの場合で，4〜710 msec となる．なお，鍵の生成は，いず

れのマシンでも 1 個の鍵の生成あたり 1 msec 以下の処理時間となっているため，ほとんど影響はない．

7. おわりに

本稿では，グリッド環境で適応的な動作を行うモバイルエージェントシステム，Ja-Net on Grid におけるモバイルエージェントのセキュリティに注目し，CE がタスクを実行する ACERE からの攻撃に対処する移動手法を提案し，その詳細について述べた．本手法により，悪意のある ACERE が Grid ネットワークに存在した場合でも，その ACERE を避けることなく，CE は安全にタスクを遂行できる．

今後は本手法を適用する際のコストをさらに詳しく評価し，実アプリケーションに適用した際にパフォーマンスが低下しないことを確認していく予定である．また Ja-Net on Grid では，あるタスクから派生したサブタスクにおいて最終的な結果にどれだけ寄与するかといった重要性が異なる場合に，それぞれのサブタスクの重要性を考慮しつつ，計算リソースを効率的に使用することを目的としている [11]．今後は，これらのタスクの重要性を ACERE の信頼度やパフォーマンスの違いとともに考慮し，より効率的にタスク処理が行えるように Ja-Net on Grid を拡張していく予定である．

謝辞 本稿を執筆するにあたってご協力を頂いた西尾研究室の諸氏に感謝する．なお，本研究の一部は，文部科学省 21 世紀 COE プログラム「ネットワーク共生環境を築く情報技術の創出」，文部科学省特定研究領域 (C)「Grid 技術を適応した新しい研究手法とデータ管理技術の研究」（プロジェクト番号:13224059），ならびに，科学研究費補助金 (基盤研究 (B)(2))「大規模な仮想空間システムを構築する放送型サイバースペースに関する研究」（プロジェクト番号:15300033）によっている．ここに記して謝意を表す．

参考文献

1) Bennet, S. Y.: A sanctuary for mobile agents, Secure Internet programming: security issues for mobile and distributed objects, Springer-Verlag (2001).
2) Giovanni, V.: Protecting Mobile Agents through Tracing, Proceedings of the 3rd ECOOP Workshop on Mobile Object Systems, Jyvälskylä, Finland (1997).
3) Jansen, W.: Countermeasures for Mobile Agent Security, Computer Communications, Special Issue on Advanced Security Techniques for Network Protection, Elsevier Science BV (2000).
4) Sander,T. and Tschudin, C.F.: Protecting Mobile Agents Against Malicious Hosts, Mobile Agents and Security, Lecture Notes in Computer Science, Vol.1419, Springer-Verlag, pp.44-60 (1998).
5) SETI@home: Search for Extraterrestrial Intelligence at home, http://setiathome.ssl.berkeley.edu/.
6) Sun Microsystems: AccessibleObject, http://java.sun.com/j2se/1.4.2/docs/api/java/lang/reflect/AccessibleObject.html.
7) The Globus Project, http://www.globus.org/.
8) Yariv, A. and Danny, B. L.: Agent Design Patterns: Elements of Agent Application Design, Autonomous Agents 98 (1998).
9) 板生知子, 塚本昌彦, 山本淳, 田中聡: 高度ユビキタス環境のための Ja-Net アーキテクチャ, 電子情報通信学会総合大会論文集 (2004).
10) 中田秀基: グリッドコンピューティング－ Globus, Java CoG キットによるグリッドポータル構築, JAVA PRESS, Vol.27 (2002).
11) 沼田哲史, 板生知子, 小川剛史, 塚本昌彦, 西尾章治郎: 動的なグリッド環境における効率的でセキュアなリソース利用のためのモバイルエージェントシステム Ja-Net on Grid, 情報処理学会論文誌「データベース」(TOD 24) (2004).